

Self Stabilizing Virtual Synchrony*

Shlomi Dolev[†] Chryssis Georgiou[‡] Ioannis Marcoullis[‡]
 Elad M. Schiller[§]

Abstract

Virtual synchrony is an important abstraction that is proven to be extremely useful when implemented over asynchronous, typically large, message-passing distributed systems. Fault tolerant design is a key criterion for the success of such implementations. This is because large distributed systems can be highly available as long as they do not depend on the full operational status of every system participant. Namely, they employ redundancy in numbers to overcome non-optimal behavior of participants and to gain global robustness and high availability.

Self-stabilizing systems can tolerate transient faults that drive the system to an arbitrary unpredicted configuration. Such systems automatically regain consistency from any such arbitrary configuration, and then produce the desired system behavior. Practically self-stabilizing systems ensure the desired system behavior for practically infinite number of successive steps e.g., 2^{64} steps.

We present the first practically self-stabilizing virtual synchrony algorithm. The algorithm is a combination of several new techniques that may be of independent interest. In particular, we present a new counter algorithm that establishes an efficient practically unbounded counter, that in turn can be directly used to implement a self-stabilizing Multiple-Writer Multiple-Reader (MWMR) register emulation. Other components include self-stabilizing group membership, self-stabilizing multicast, and self-stabilizing emulation of replicated state machine. As we base the replicated state machine implementation on virtual synchrony, rather than consensus, the system progresses in more extreme asynchronous executions in relation to consensus-based replicated state machine.

*A preliminary version of this work has appeared in the proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (*SSS'15*).

[†]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel. Email dolev@cs.bgu.ac.il Partially supported by Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences and Israel Science Foundation (grant number 428/11).

[‡]Department of Computer Science, University of Cyprus, Nicosia, Cyprus. Email {chryssis, imarco01}@cs.ucy.ac.cy. Supported by the University of Cyprus.

[§]Department of Engineering and Computer Science, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden, Email elad@chalmers.se.

1 Introduction

Virtual Synchrony (VS) has been proven to be very important in the scope of fault-tolerant distributed systems [5]. The VS property ensures that two or more processors that participate in two consecutive communicating groups should have delivered the same messages. Systems that support the VS abstraction are designed to operate in the presence of fail-stop failures of a minority of the participants. Such a design fits large computer clusters, datacenters and cloud computing, where at any given time some of the processing units are non-operational. Systems that cannot tolerate such failures degrade their functionality and availability to the degree of unuseful systems.

Group communication systems that realize the VS abstraction provide services, such as *group membership* and *reliable group multicast*. The group membership service is responsible for providing the current *group view* of the recently live and connected group members, i.e., a processor set and a unique *view identifier*, which is a sequence number of the view installation. The reliable group multicast allows the service clients to exchange messages with the group members as if it was a single communication endpoint with a single network address and to which messages are delivered in an atomic fashion, thus any message is either delivered to all recently live and connected group members prior to the next message, or is not delivered to any member. The challenges related to VS consist of the need to maintain atomic message delivery in the presence of asynchrony and crash failures. VS facilitates the implementation of a replicated state machine [5] that is more efficient than classical consensus-based implementations that start every multicast round with an agreement on the set of recently live and connected processors. It is also usually easier to implement [5]. To the best of our knowledge, no *self-stabilizing virtual synchrony* solution exists.

Transient violations of design assumptions can lead a system to an arbitrary state. For example, the assumption that error detection ensures the arrival of correct messages and the discarding of corrupted messages, might be violated since error detection is a probabilistic mechanism that may not detect a corrupt message. As a result, the message can be regarded as legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, requiring human intervention. In the presence of transient faults, large multicomputer systems providing VS-based services can prove hard to manage and control. One key problem, not restricted to virtually synchronous systems, is catering for counters (such as view identifiers) reaching an arbitrary value. How can we deal with the fact that transient faults may force counters to wrap around to the zero value and violate important system assumptions and correctness invariants, such as the ordering of events? A self-stabilizing algorithm [10] can automatically recover from such unexpected failures, possibly as part of after-disaster recovery or even after benign temporal violations of the assumptions made in the design of the system. We tackle this issue in our work.

Contributions. We present the first self-stabilizing virtual synchrony solution. Specifically:

- We provide a self-stabilizing counter algorithm using bounded memory and communication bandwidth, and yet (many writers) can increment the counter for an unbounded number of times in the presence of processor crashes and unbounded communication delays.
- Our counter algorithm is modular with a simple interface for increasing and reading the counter, as well as providing the identifier of the processor that has incremented it.
- At the heart of our counter algorithm is the underlying labeling algorithm that extends the label scheme of Alon et al. [1] to support multiple writers, whilst the algorithm specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to “discover” the greatest label and discard all obsolete ones.
- An immediate application of our counter algorithm is a self-stabilizing MWMR register emulation.
- The self-stabilizing counter algorithm, together with the proposed implementations of a self-stabilizing reliable multicast service and membership service, are composed to yield a self-stabilizing VS-based State Machine Replication (SMR) implementation.

Related Work. Leslie Lamport was the first to introduce SMR, presenting it as an example in [17]. Schneider [20] gave a more generalized approach to the design and implementation of SMR protocols. Group communication services can implement SMR by providing reliable multicast that guarantees VS [4]. Birman et al. were the first to present VS and a series of improvements in the efficiency of ordering protocols [6]. Birman gives a concise account of the evolution of the VS model for SMR in [5].

Research during the last recent decades resulted in an extensive literature on ways to implement VS and SMR, as well as industrial construction of such systems. A recent research line on (practically) self-stabilizing versions of replicated state machines [1, 9, 13, 14] obtains self-stabilizing replicated state machines in shared memory as well as in synchronous and asynchronous message passing systems.

The bounded labeling scheme and the use of practically unbounded sequence numbers proposed in [1], allow the creation of self-stabilizing bounded-size solutions to the never-exhausted counter problem in the restricted case of a single writer. In [9] a self-stabilizing version of Paxos was developed that led to a self-stabilizing consensus-based SMR implementation. To this end, they extend the labeling scheme of [1] to allow for multiple counter writers, since unbounded counters are required for ballot numbers. Extracting this scheme for other uses does not seem intuitive. We present a simpler and significantly more communication efficient self-stabilizing (bounded-size never-exhausted) counter that also supports many writers, where a single label rather than a vector of labels needs to be communicated. Our solution is *highly modular* and can be easily used in any similar setting requiring such counters.

Practically-stabilizing VS and self-stabilizing VS are identical when VS is defined by the behaviour of classical VS algorithms that use (bounded) counters. These algorithms preserve the VS requirements as long as the counters do not reach their upper bound. In our setting, if a counter reaches the upper bound due to a transient fault our self-stabilizing/practically-stabilizing solution introduces a new epoch with new sequence numbers. It, thus, converges to act exactly as the non-stabilizing VS (for the same number of steps) as an initialized non-stabilizing VS algorithm.

Next, in Section 2, we overview our construction, describing the core techniques and the way they establish the desired properties. In Section 3 we present the model of computation we consider. Section 4 details the self-stabilizing Labeling and Increment Counter algorithms. In Section 5 we detail the self-stabilizing Virtual Synchrony algorithm and the resulting replicate state machine emulation. We conclude in Section 6.

2 Our Results in a Nutshell

We start with the necessary succinct description of the system settings (more details in Section 3). We consider an asynchronous message passing system consisting of n communicating processors; each with a unique identifier. We assume that up to a minority of the processors might become inactive. The communication network topology is of a fully connected graph. Any message that is sent infinitely often from one active processor to another active processor is eventually received. We often use the term *packets* for low-level messages, distinguishing packets that are retransmitted to ensure delivery of high-level messages exactly once. Moreover, we assume that the communication links have known bounded capacity, and thus we can use existing self-stabilizing data-link layer algorithms for emulating reliable FIFO communication channel protocols that can even tolerate message omission, duplication as well as transient faults [11, 12].

2.1 Bounded labeling scheme for multiple writers

As mentioned, Alon et al. [1] presented a bounded labeling scheme to implement an SWMR register emulation in a message-passing system. The *labels* (also called *epochs*) allow the system to stabilize, since once a label is established, the integer counter related to this label is considered to be practically infinite, as a 64-bit integer is practically infinite and sufficient for the lifespan of any reasonable system. We extend the labeling scheme of [1] to support multiple writers, by including the epoch creator (writer) identity to break symmetry, and decide which epoch is the most recent one, even when two or more creators concurrently create a new label.

When all processors (and hence potential writers) are active, the scheme can be viewed as a simple extension of the one of [1]. Informally speaking, the scheme assures that each processor p_i eventually “cleans up” the system from obsolete labels of which p_i appears to be the creator (for example, such

labels could be present in the system’s initial arbitrary state). Specifically, p_i maintains a bounded FIFO history of such labels that it has recently learned, while communicating with the other processors, and creates a label greater than all that are in its history; call this p_i ’s *local maximal label*. In addition, each processor seeks to learn the *globally maximal label*, that is, the label in the system that is the greatest among the local maximal ones. Unfortunately, when some processors are not active, finding a global maximal becomes challenging, since these processors will not “clean up” their local labels. So, roughly speaking, the active processors need to do this indirectly without knowing which processors are inactive. To overcome this problem, we have each processor maintaining bounded FIFO histories on labels appearing to have been created by other processors. These histories eventually accumulate the obsolete labels of the inactive processors. We show that even in the presence of (a minority of) inactive processors, starting from an arbitrary state, the system eventually converges to use a global maximal label.

Let us explain why obsolete labels from inactive processors can create a problem when no one ever cleans (cancels) them up. Consider a system starting in a state that includes a cycle of labels $\ell_1 \prec \ell_2 \prec \ell_3 \prec \ell_1$, all of the same creator, say p_x , where \prec is the label order relation. If p_x is active, it will eventually learn about these labels and introduce a label greater than them all. But if p_x is inactive, the system’s asynchronous nature may cause a repeated cyclic label adoption, especially when p_x has the greatest processor identifier, as these identifiers are used to break symmetry. Say that an active processor learns and adopts ℓ_1 as its global maximal label. Then, it learns about ℓ_2 and hence adopts it, while forgetting about ℓ_1 . Then, learning of ℓ_3 it adopts it. Lastly, it learns about ℓ_1 , and as it is greater than ℓ_3 , it adopts ℓ_1 once more, as the greatest in the system; this can continue indefinitely. By using the bounded FIFO histories, such labels will be accumulated in the histories and hence will not be adopted again, ending this vicious cycle.

2.2 Practically infinite counter for multiple writers

Using our labeling scheme, we show how to implement a practically infinite counter supporting multiple writers. The idea is to extend the labeling scheme to handle *counters*, where a counter consists of a *label*, as used in the labeling scheme; an integer *sequence number*, ranging from 0 to 2^b , where b is large enough, say $b = 64$; and a processor *id*. Conceptually, if the system stabilizes to use a global maximal label, then the pair of the sequence number and the processor id (of this sequence number) can be used as an unbounded counter, as used, for example, in MWMR register implementations [18, 19]. Specifically, we say that counter $cnt_1 = \langle \ell_1, seqn_1, wid_1 \rangle$ is *smaller* than counter $cnt_2 = \langle \ell_2, seqn_2, wid_2 \rangle$ if $(\ell_1 \prec \ell_2)$ or $((\ell_1 = \ell_2) \text{ and } (seqn_1 < seqn_2))$ or $((\ell_1 = \ell_2) \text{ and } (seqn_1 = seqn_2) \text{ and } (wid_1 < wid_2))$. Note that when processors have the same label, the above relation forms a total ordering and processors can increment a shared counter also when attempting to do so concurrently. We argue that starting from any initial configuration, eventually the counter

algorithm supports such increments.

The counter increment algorithm uses the same structures and procedures as the labeling algorithm, but now with counters instead of labels. To increment the counter, a processor p_i first sends a request to all other processors querying the counter they consider as the global maximum and awaits for responses from a majority. Using a similar procedure as the labeling algorithm it (eventually) finds the maximal epoch label and the maximal sequence number it knows for this label. In other words, it collects counters and finds the counter(s) with the largest global label; there can be more than one such counter, in which case it returns the one with the highest sequence number, breaking symmetry with the sequence number processor identifiers. Then it checks whether this maximal sequence number is *exhausted*, that is, the sequence number is equal or larger than 2^{64} (this could be, for example, due to the arbitrary values in the configuration the system starts in). When this is the case, it proceeds to find a new maximal label until it finds one that is not exhausted and uses the maximal sequence number it knows for this epoch label. Then the processor increments the sequence number by one, sets its identifier as the writer of the sequence number and sends the new counter to all processors, and awaits for acknowledgment from a majority (this is, in spirit, similar to the two-phase write operation of MWMR register implementations, focusing on the sequence number rather than on an associated value).

Note that when a processor p_i establishes a new label ℓ as the global maximum, it sets the corresponding counter $cnt = \langle \ell, 0, i \rangle$; in this case, the label creator identifier and the sequence number writer identifier is i . When there is an already established maximal label ℓ in the system and processor p_i wants to increment the counter, it increases the corresponding (to ℓ) maximal sequence number found ($maxseqn$) by one, and sets the counter $cnt = \langle \ell, maxseqn + 1, i \rangle$; in this case, it is possible that the label creator identifier and the sequence number writer identifier are not the same, i.e., if p_i was not the creator of label ℓ . Also, note that some extra care is needed with respect to counter bookkeeping so as not to increase the size of the bounded histories used in the labeling algorithm. Having a counter increment algorithm, it is not difficult to obtain a practically self-stabilizing MWMR register implementation; counters are associated with values and the counter increment algorithm is run with this small amendment (more details in Sect. 4.3).

2.3 Practically self-stabilizing virtual synchrony and Replicated state machine

Our self-stabilizing Virtual Synchrony implementation combines the implementation of the our counter algorithm and a self-stabilizing FIFO data link between any two participants; the latter is used to implement a self-stabilizing reliable multicast service and a self-stabilizing failure detector (used for the membership service).

Data link implementation. One version of a self-stabilizing FIFO data link implementation that we can use, is based on the fact that communication links have bounded capacity. Packets are retransmitted until more than the total capacity acknowledgments arrive; while acknowledgments are sent only when a packet arrives (not spontaneously) [11, 12]. Over this data-link, the two connected processors can constantly exchange a “token”. Specifically, the sender (possibly the processor with the highest identifier among the two) constantly sends packet π_1 until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π_2 , and so on and so forth. This assures that the receiver has received packet π_1 before the sender starts sending packet π_2 . This can be viewed as a token exchange. We use the abstraction of the token carrying messages back and forth between any two communication entities. We use this token exchange technique when implementing a reliable multicast procedure, as well as a the basis for a *heartbeat* for detecting whether a processor is active or not; when a processor is no longer active, the token will not be returned back to the other processor.

Reliable multicast implementation. As we will see next, we use a coordinator to exchange messages (by multicasting) within the group. The coordinator requests, collects and combines input from the group members, and then it multicasts the updated information. Specifically, when the coordinator decides to collect inputs, it waits for the token to arrive from each group participant. Whenever a token arrives from a participant, the coordinator uses the token to send the request for input to that participant, and waits the token to return with some input (possibly \perp , when the participant does not have a new input). Once the coordinator receives an input from a certain participant with respect to this multicast invocation, the corresponding token will not carry any new requests to receive input from the same participant; of course, the tokens continue to move back and forth. When all inputs are received, the processor combines them and again uses the token to carry the updated information. Once this is done, the coordinator can proceed to the next input collection, when needed.

Failure detector implementation. Every processor p maintains a heartbeat integer counter for every other processor q . Whenever processor p receives the token from processor q over their data link, processor p resets the counter’s value to zero and increments all the integer counters associated with the other processors by one, up to a predefined threshold value W . Once the heartbeat counter value of a processor q reaches W , the failure detector of processor p considers q as inactive. In other words, the failure detector at processor p considers processor q to be active, if and only if the heartbeat associated with q is strictly less than W . This is essentially the failure detector mentioned in [9]. Note that for the correctness of our virtual synchrony algorithm, we require a weaker failure detector. Specifically, we require that at least one processor is not suspected, for sufficiently long time, only by a majority of the processors, as opposed to an eventually perfect failure detector that ensures that after a

certain time, no active processor suspects any other active processor.

Self-stabilizing virtual synchrony implementation. The algorithm is coordinator-based and we consider a *primary-group* implementation [6]. To assign view identifiers, we use our counter increment algorithm. Specifically, the view identifier is a triple that includes an epoch (label), the currently highest counter, cnt , which the counter algorithm obtains, and the processor that has created this counter, $cnt.wid$ (writer), which is also the view coordinator. Note that this defines a simple interface with the counter algorithm, which provides an identical output. Furthermore, the view membership uses the output of the coordinator’s failure detector for defining the set of view members; this helps to maintain a consistent membership among the group members, despite inaccuracies between the various failure detectors; as we show, this does not break the virtual synchrony property, as long as the majority-based failure detector property is preserved. Recall that the coordinator is responsible for the consistency of the multicast mechanism within the group.

It may happen that the system reaches a configuration with no coordinator. For example, this could be the case in the arbitrary configuration that the system starts in, or in the case that the coordinator of an installed view is no longer active. Each participant that detects that it has no coordinator, seeks for potential candidates based on the exchanged information. A processor p regards a processor q as a candidate, if q is active according to p ’s failure detector, and there is a majority of processors that also think so (all these are based on p ’s knowledge, which due to asynchrony might not be up to date). When there is more than one such candidate, processor p checks whether there is a candidate that has proposed a higher counter among the candidates. If there is one, then p considers it to be the coordinator and waits to hear from it (or learn that it is not active). If there is none, and based on its knowledge there is a majority of processors that also do not have a coordinator, then processor p acquires a counter from the counter increment algorithm and proposes a new view, with view ID, the counter, and group membership, the set of processors that appear active according to its failure detector. As we show, if p receives an “accept” message from *all* the processors in the view, then it proceeds to install the view, unless another processor who has obtained a higher counter does so. In a transition from one view to the next, there can be several processors attempting to become the coordinator (namely, those who according to their knowledge have a supporting majority). Still, by exploiting the intersection property of the supporting majorities we prove that each of these processors will propose a view at most once, and out of these, one view will be installed (i.e., we do not have never-ending attempts for new views to be installed).

The *virtual synchrony property* essentially requires that any two processors that participate in two consecutive groups should have delivered the same messages. Roughly speaking, our algorithm preserves this property as follows: Once a processor does not have a coordinator, it stops participating in group multicasting, and prior to delivering a new multicast message in a new view, the

algorithm assures that the coordinator of this new view has collected all the participants' last delivered messages (in their prior views) and resends the messages appearing not to have been delivered uniformly. To do so, each participant keeps the last delivered message and the view identifier that delivered this message. We show that this, together with the intersection property of majorities, (and after taking care of some subtle issues,) provides the virtual synchrony property. Starting from an arbitrary configuration, we show that if there is no valid coordinator, eventually a processor proposes a new view and, therefore, a valid coordinator is eventually elected. To assure this, processors continuously exchange through the failure detector's token their coordinator's identifier (or \perp if there's no such). This helps to detect initially corrupted states when, say a processor p_i might consider p_j as its coordinator, but p_j does not consider itself to be the coordinator. Combining the above with the self-stabilization of the counter increment algorithm, the data links, the failure detector and multicast, we are able to guarantee reaching a legal execution in which the virtual synchrony property is always satisfied.

Self-stabilizing replicate state machine implementation. Each participant maintains a replica of the state machine and the last processed (composite) message. Note that we bound the memory used to store the history of the replicated state machine by deciding to have the (encapsulated influence of the history represented by the) current state of the replicated state machine. In addition, each participant maintains the last delivered (composite) message to ensure common reliable multicast, as the coordinator may stop being active prior to ensuring that all members received a copy of the last multicast message. Whenever a new coordinator is elected, the coordinator inquires all members (forming a majority) for the most updated state and delivered message. Since at least one of the members, say p_i , participated in the group in which the last completed state machine transition took place, p_i 's information will be recognized as associated with the largest counter, adopted by the coordinator that will in turn, assign the most updated state and available delivered message to all the current group members, in essence satisfying the virtual synchrony property. Then the coordinator, as part of the multicast procedure, collects inputs received from the environment before ensuring that all group members apply these inputs to the replica state machine. Note that the received multicast message consists of input (possibly \perp) from each of the processors, thus, the processors need to apply one input at a time, the processors may apply them in an agreed upon sequential order, say from the input of the first processor to the last. Alternatively, the coordinator may request one input at a time in a round-robin fashion and multicast it. Finally, to ensure that the system stabilizes when started in an arbitrary configuration, every so often, the coordinator assigns the state of its replica to the other members.

Perhaps some of the above ideas appear conceptually clear, however, there are low-level critical details that are essential to realizing them and prove them correct, as we are ready to describe.

3 System Settings

We consider an asynchronous message passing system as the one used in [1]. The system includes a set P of n communicating processors; we refer to the processor with identifier i , as p_i . We assume that up to a minority of processors may become inactive. We assume that the system runs on top of a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels [11, 12]. The network topology is of a fully connected graph where every two processors exchange (low-level messages called) *packets* to enable a reliable delivery of (high level) messages. When no confusion is possible we use the term messages for packets. The communication links have bounded capacity, so that the number of packets in every given instance is bounded by a constant. When processor p_i sends a packet, pkt , to processor p_j , the operation *send* inserts a copy of pkt to the FIFO queue that represents the communication channel from p_i to p_j , while respecting an upper bound on the number of packets in the channel, possibly omitting the new packet or one of the already sent packets. When p_j receives pkt from p_i , pkt is dequeued from the queue representing the channel. We assume that packets can be spontaneously omitted (lost) from the channel, however, a packet that is sent infinitely often is received infinitely often.

The code of self-stabilizing algorithms usually consists of a do forever loop that contains communication operations with the neighbors and validation that the system is in a consistent state as part of the transition decision. An *iteration* is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters branches).

Every processor, p_i , executes a program that is a sequence of (*atomic*) *steps*, where a step starts with local computations and ends with a single communication operation, which is either *send* or *receive* of a packet. For ease of description, we assume the interleaving model, where steps are executed atomically, a single step at any given time. An input event can be either the receipt of a packet or a periodic timer triggering p_i to (re)send. Note that the system is asynchronous and the rate of the timer is totally unknown.

The *state*, s_i , of a node p_i consists of the value of all the variables of the node including the set of all incoming communication channels. The execution of an algorithm step can change the node's state. The term (*system*) *configuration* is used for a tuple of the form (s_1, s_2, \dots, s_n) , where each s_i is the state of node p_i (including messages in transit for p_i). We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \dots$ as an alternating sequence of system configurations c_x and steps a_x , such that each configuration c_{x+1} , except the initial configuration c_0 , is obtained from the preceding configuration c_x by the execution of the step a_x . A practically infinite execution is an execution with many steps (and iterations), where many is defined to be proportional to the time it takes to execute a step and the life-span time of a system.

We define the system's task by a set of executions called *legal executions* (*LE*) in which the task's requirements hold, we use the term *safe configuration* for any configuration in *LE*. An algorithm is *self-stabilizing* with relation to

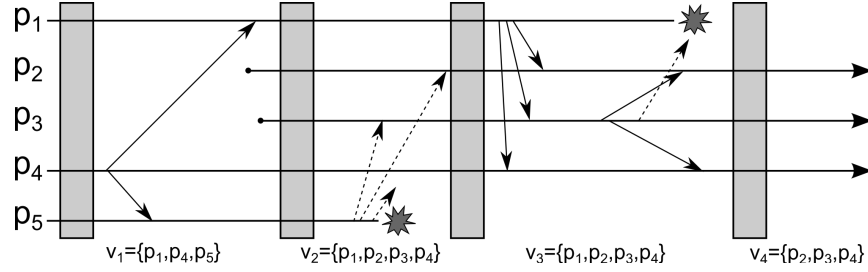


Figure 1: An execution satisfying the VS property. The grey boxes indicate a new view installation, and the example shows four views. View v_1 initially with membership $\{p_1, p_4, p_5\}$. The reliable multicast reaches all members of the group. Two new processors p_2 and p_3 join the group, forming view v_2 . In this view, p_5 crashes before completing its multicast which is ignored (dashed lines). The new view v_3 is formed to exclude p_5 , and in it, p_1 manages a successful multicast before crashing. The multicast of p_3 is reliable and guaranteed to be delivered to all non-crashed within the view, that is excluding p_1 which might or might not have received it (dotted line). A new view is then formed to encapture the failure of p_1 .

the task LE when every (unbounded) execution of the algorithm reaches a safe configuration with relation to the algorithm and the task. An algorithm is *practically stabilizing* with relation to the task LE if in any practically infinite execution a safe configuration is reached.

The *virtual synchrony task* requires that any two processors that share the same sequence of views, ought to *deliver* the same identical message sets in these views. The legal execution of virtual synchrony is defined in terms of the input and output sequences of the system with the environment. When a majority of processors are continuously active every external input (and only the external inputs) should be atomically accepted and processed by the majority of the active processors. Note that there is no delivery and processing guarantee in executions in which there is no majority, still in these executions any delivery and processing is due to a received environment input. An exemplar virtually synchronous execution can be found in Figure 1.

Notation. Throughout the paper we use the following notation. Let y and y' be two objects that both include the field x . We denote $(y =_x y') \equiv (y.x = y'.x)$.

4 Self-stabilizing Labeling Scheme and Counter Algorithm

In this section, we first present and prove correct of the proposed self-stabilizing labeling algorithm and then explain how this can be extended to implement self-stabilizing practically unbounded counters in Section 4.3.

Algorithm 1: The *nextLabel()* function; code for p_i

```

1 For any non-empty set  $X \subseteq D$ , function  $pick(d, X)$  returns  $d$  arbitrary elements
  of  $X$ ;
   input :  $S = \langle \ell_1, \ell_2 \dots, \ell_k \rangle$  set of  $k$  labels.
   output :  $\langle i, newSting, newAntistings \rangle$ 
2 let  $newAntistings = \{\ell_j.sting : \ell_j \in S\}$ ;
3  $newAntistings \leftarrow newAntistings \cup pick(k - |newAntistings|,$ 
   $D \setminus newAntistings)$ ;
4 return
   $\langle i, pick(1, D \setminus (newAntistings \cup \{\cup_{\ell_j \in S} \ell_j.Antistings\})), newAntistings \rangle$ 

```

4.1 Labeling Algorithm for Concurrent Label Creations

4.1.1 Bounded Labeling Scheme

We extend the labeling scheme of [1] to support wait-free multi-writer systems. We do so, by extending the label with a *label creator's* identifier, so as to break symmetry and decide about the most recent epoch even when two or more writers concurrently attempt to create a new label.

Specifically, we consider the set of integers $D = [1, k^2 + 1]$. A *label* (or *epoch*) is a triple $\langle lCreator, sting, Antistings \rangle$, where $lCreator$ is the identity of the processor that established (created) the label, $Antistings \subset D$ with $|Antistings| = k$, and $sting \in D$. Given two labels ℓ_i, ℓ_j , we define the relation $\ell_i \prec_{lb} \ell_j \equiv (\ell_i.lCreator < \ell_j.lCreator) \vee (\ell_i.lCreator = \ell_j.lCreator \wedge ((\ell_i.sting \in \ell_j.Antistings) \wedge (\ell_j.sting \notin \ell_i.Antistings)))$; we use $=_{lb}$ to say that the labels are identical. Note that the relation \prec_{lb} does not define a total order. For example, when $\ell_i =_{lCreator} \ell_j$ and $(\ell_i.sting \notin \ell_j.Antistings)$ and $(\ell_j.sting \notin \ell_i.Antistings)$ these labels are *incomparable*. As in [1], we demonstrate that one can still use this labeling scheme as long as it is ensured that eventually a label greater than all other labels in the system is introduced. We say that a label ℓ **cancel**s another label ℓ' , either if they are incomparable or they have the same $lCreator$ but ℓ is greater than ℓ' (with respect to *sting* and *Antistings*). A label with creator p_i is said to belong to p_i 's domain.

Function *nextLabel()*, Algorithm 1, gets a set of at most k labels as input and returns a new label that is greater than all of the labels of the input. It has the same functionality as the function called *Next_b()* in [1], but it additionally considers the label creator. The function essentially composes a new *Antistings* set from the stings of all the labels it has as input, and chooses a *sting* that is in none of the *Antistings* of the input labels. In this way it ensures that the new label is greater than any of the input. Note that the function takes k *Antistings* of k labels that are not necessarily distinct, implying at most k^2 distinct integers and thus the choice of $|D| = k^2 + 1$ allows to always obtain a greater integer as the *sting*.

4.1.2 The Labeling Algorithm

The labeling algorithm (Algorithm 2) specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to “discover” their greatest label and cancel all obsolete ones. As we will be using pairs of labels with the *same* label creator, for the ease of presentation, we will be referring to these two variables as the *(label) pair*. The first label in a pair is called *ml*. The second label is called *cl* and it is either \perp , or equal to a label that cancels *ml* (i.e., *cl* indicates whether *ml* is an obsolete label or not).

The processor state. Each processor stores an array of label pairs, $max_i[n]$, where $max_i[i]$ refers to p_i ’s maximal label pair and $max_i[j]$ considers the most recent value that p_i knows about p_j ’s pair. Processor p_i also stores the pairs of the most-recently-used labels in the array of queues $storedLabels_i[n]$. The j -th entry refers to the queue with pairs from p_j ’s domain, i.e., that were created by p_j . The algorithm makes sure that $storedLabels_i[j]$ includes only label pairs with unique *ml* from p_j ’s domain and that at most one of them is *legitimate*, i.e., not canceled. Queues $storedLabels_i[j]$ for $i \neq j$, have size $n + m$ whilst $storedLabels_i[i]$ has size $2(mn + 2n^2 - 2n)$ where m is the system’s total link capacity in labels. We later show (c.f. Lemmas 4.3 and 4.4) that these queue sizes are sufficient to prevent overflows of useful labels.

Information exchange between processors. Processor p_i takes a step whenever it receives two pairs $\langle sentMax, lastSent \rangle$ from some other processor. We note that in a legal execution p_j ’s pair includes both *sentMax*, which refers to p_j ’s maximal label pair $max_j[j]$, and *lastSent*, which refers to a recent label pair that p_j received from p_i about p_i ’s maximal label, $max_j[i]$ (line 16).

Whenever a processor p_j sends a pair $\langle sentMax, lastSent \rangle$ to p_i , this processor stores the value of the arriving *sentMax* field in $max_i[j]$ (line 19). However, p_j may have local knowledge of a label from p_i ’s domain that cancels p_i ’s maximal label, *ml*, of the last received *sentMax* from p_i to p_j that was stored in $max_j[i]$. Then p_j needs to communicate this canceling label in its next communication to p_i . To this end, p_j assigns this canceling label to $max_j[i].cl$ which stops being \perp . Then p_j transmits $max_j[i]$ to p_i as a *lastSent* label pair, and this satisfies $lastSent.cl \not\leq_b lastSent.ml$, i.e., *lastSent.cl* is either greater or incomparable to *lastSent.ml*. This makes *lastSent* illegitimate and in case this still refers to p_i ’s current maximal label, p_i must cancel $max_i[i]$ by assigning it with *lastSent* (and thus $max_i[i].cl = lastSent.cl$) as done in line 20. Processor p_i then processes the two pairs received (lines 21 to 28).

Label processing. Processor p_i takes a step whenever it receives a new pair message $\langle sentMax, lastSent \rangle$ from processor p_j (line 17). Each such step starts by removing *stale* information, i.e., misplaced or doubly represented labels (line 9). In the case that stale information exists, the algorithm empties the entire label storage. Processor p_i then tests whether the arriving two pairs

Algorithm 2: Self-Stabilizing Labeling Algorithm; code for p_i

```

1 Variables:
2  $max[n]$  of  $\langle ml, cl \rangle$ :  $max[i]$  is  $p_i$ 's largest label pair,  $max[j]$  refers to  $p_j$ 's label pair
   (canceled when  $max[j].cl \neq \perp$ ).
3  $storedLabels[n]$ : an array of queues of the most-recently-used label pairs, where
    $storedLabels[j]$  holds the labels created by  $p_j \in P$ . For  $p_j \in (P \setminus \{p_i\})$ ,  $storedLabels[j]$ 's
   queue size is limited to  $(n + m)$  w.r.t. label pairs, where  $n = |P|$  is the number of processors
   in the system and  $m$  is the maximum number of label pairs that can be in transit in the
   system. The  $storedLabels[i]$ 's queue size is limited to  $(n(n^2 + m))$  pairs. The operator
    $add(\ell)$  adds  $lp$  to the front of the queue, and  $emptyAllQueues()$  clears all  $storedLabels[]$ 
   queues. We use  $lp.remove()$  for removing the record  $lp \in storedLabels[]$ . Note that an
   element is brought to the queue front every time this element is accessed in the queue.
4 Notation: Let  $y$  and  $y'$  be two records that include the field  $x$ . We denote  $y =_x y' \equiv (y.x$ 
    $= y'.x)$ 
5 Macros:
6  $legit(lp) = (lp = \langle \bullet, \perp \rangle)$ 
7  $labels(lp) : \text{return } (storedLabels[lp.ml.lCreator])$ 
8  $double(j, lp) = (\exists lp' \in storedLabels[j] : ((lp \neq lp') \wedge ((lp =_{ml} lp') \vee (legit(lp) \wedge legit(lp')))))$ 
9  $staleInfo() = (\exists p_j \in P, lp \in storedLabels[j] : (lp \neq_{lCreator} j) \vee double(j, lp))$ 
10  $recordDoesntExist(j) = (\langle max[j].ml, \bullet \rangle \notin labels(max[j]))$ 
11  $notgeq(j, lp) = \text{if } (\exists lp' \in storedLabels[j] : (lp'.ml \not\leq_{lb} lp.ml)) \text{ then return}(lp'.ml)$ 
   else return( $\perp$ )
12  $canceled(lp) = \text{if } (\exists lp' \in labels(lp) : ((lp' =_{ml} lp) \wedge \neg legit(lp')))$  then return( $lp'$ )
   else return( $\perp$ )
13  $needsUpdate(j) = (\neg legit(max[j]) \wedge \langle max[j].ml, \perp \rangle \in labels(max[j]))$ 
14  $legitLabels() = \{max[j].ml : \exists p_j \in P \wedge legit(max[j])\}$ 
15  $useOwnLabel() = \text{if } (\exists lp \in storedLabels[i] : legit(lp))$  then  $max[i] \leftarrow lp$ 
   else  $storedLabels[i].add(max[i] \leftarrow \langle nextLabel(), \perp \rangle)$  // For every
    $lp \in storedLabels[i]$ , we pass in  $nextLabel()$  both  $lp.ml$  and  $lp.cl$ .
16 upon  $transmitReady(p_j \in P \setminus \{p_i\})$  do  $transmit(\langle max[i], max[j] \rangle)$ 
17 upon  $receive(\langle sentMax, lastSent \rangle)$  from  $p_j$ 
18 begin
19    $max[j] \leftarrow sentMax$ ;
20   if  $\neg legit(lastSent) \wedge max[i] =_{ml} lastSent$  then  $max[i] \leftarrow lastSent$ 
21   if  $staleInfo()$  then  $storedLabels.emptyAllQueues()$ 
22   foreach  $p_j \in P : recordDoesntExist(j)$  do  $labels(max[j]).add(max[j])$ 
23   foreach  $p_j \in P, lp \in storedLabels[j] : (legit(lp) \wedge (notgeq(j, lp) \neq \perp))$  do
      $lp.cl \leftarrow notgeq(j, lp)$ 
24   foreach  $p_j \in P, lp \in labels(max[j]) : (\neg legit(max[j]) \wedge (max[j] =_{ml} lp) \wedge legit(lp))$  do
      $lp \leftarrow max[j]$ 
25   foreach  $p_j \in P, lp \in storedLabels[j] : double(j, lp)$  do  $lp.remove()$ 
26   foreach  $p_j \in P : (legit(max[j]) \wedge (canceled(max[j]) \neq \perp))$  do
      $max[j] \leftarrow canceled(max[j])$ 
27   if  $legitLabels() \neq \emptyset$  then  $max[i] \leftarrow \langle \max_{\prec_{lb}}(legitLabels()), \perp \rangle$ 
28   else  $useOwnLabel()$ 

```

are already included in the label storage ($storedLabels[]$), otherwise it includes them (line 22). The algorithm continues to see whether, based on the new pairs added to the label storage, it is possible to cancel a non-canceled label pair (which may well be the newly added pair). In this case, the algorithm updates the canceling field of any label pair lp (line 23) with the canceling label of a label pair lp' such that $lp'.ml \not\leq_{lb} lp.ml$ (line 23). It is implied that since the two pairs belong to the same storage queue, they have the same processor as creator. The algorithm then checks whether any pair of the $max_i[]$ array can cause canceling to a record in the label storage (line 24), and also line 25 removes any canceled records that share the same creator identifier. The test also considers the case in which the above update may cancel any arriving label in $max[j]$ and updates this entry accordingly based on stored pairs (line 26).

After this series of tests and updates, the algorithm is ready to decide upon a maximal label based on its local information. This is the \preceq_{lb} -greatest legit label pair among all the ones in $max_i[]$ (line 27). When no such legit label exists, p_i requests a legit label in its own label storage, $storedLabels_i[i]$, and if one does not exist, will create a new one if needed (line 28). This is done by passing the labels in the $storedLabels_i[i]$ queue to the $nextLabel()$ function. Note that the returned label is coupled with a \perp and the resulting label pair is added to both $max_i[i]$ and $storedLabel_i[i]$.

4.2 Correctness proof

We are now ready to show the correctness of the algorithm. We begin with a proof overview.

Overview of the proof. The proof considers a execution R of Algorithm 2 that may initiate in an arbitrary configuration (and include a processor that takes practically infinite number of steps). It starts by showing some basic facts, such as: (1) stale information is removed, i.e., $storedLabels_i[j]$ includes only unique copies of p_j 's labels, and at most one legitimate such label (Corollary 4.1), and (2) p_i either adopts or creates the \preceq_{lb} -greatest legitimate local label (Lemma 4.2). The proof then presents bounds on the number adoption steps (Lemmas 4.3 and 4.4), that define the required queue sizes to avoid label overflows.

The proof continues to show that active processors can eventually stop adopting or creating labels, by tackling individual cases where canceled or incomparable label pairs may cause a change of the local maximal label. We show that such labels eventually disappear from the system (Lemma 4.5) and thus no new labels are being adopted or created (Lemma 4.6), which then implies the existence of a global maximal label (Lemma 4.7). Namely, there is a legitimate label ℓ_{\max} , such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ that is active throughout the execution, it holds that p_i 's local maximal label $max_i[i] = \ell_{\max}$ is the \preceq_{lb} -greatest of all the labels in $max_i[]$ and there is no label pair in $storedLabels_i[j]$ that cancels ℓ_{\max} , i.e., $((max_i[j] \preceq_{lb} \ell_{\max}) \wedge ((\forall \ell \in storedLabels_i[j]) : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max}))$. We then demonstrate that, when starting from an initial arbitrary configuration, the system eventually reaches a configuration in which there is a global maximal label (Theorem 4.2).

Before we present the proof in detail, we provide some helpful definitions and notation.

Definitions. We define \mathcal{H} to be the set of all label pairs that can be in transit in the system, with $|\mathcal{H}| = m$. So in an arbitrary configuration, there can be up to m corrupted label pairs in the system's links. We also denote $\mathcal{H}_{i,j}$ as the set of label pairs that are in transit from processor p_i to processor p_j . The

number of label pairs in $\mathcal{H}_{i,j}$ obeys the link capacity bound. Recall that the data structures used (e.g., $max_i[]$, $storedLabels_i[]$, etc) store label pairs. For convenience of presentation and when clear from the context, we may refer to the ml part of the label pair as “the label”.

4.2.1 No stale information

Lemma 4.1 says that the predicate $staleInfo()$ (line 9) can only hold during the first execution of the $receive()$ event (line 17).

Lemma 4.1 *Let $p_i \in P$ be a processor for which $\neg staleInfo_i()$ (line 9) does not hold during the k -th step in R that includes the complete execution of the $receive()$ event (from line 17 to 28). Then $k = 1$.*

Proof. Since R starts in an arbitrary configuration, there could be a queue in $storedLabels_i[]$ that holds two label records from the same creator, a label that is not stored according to its creator identifier, or more than one legitimate label. Therefore, $staleInfo_i()$ might hold during the first execution of the $receive()$ event. When this is the case, the $storedLabels_i[]$ structure is emptied (line 21). During that $receive()$ event execution (and any event execution after this), p_i adds records to a queue in $storedLabels_i[]$ (according to the creator identifier) only after checking whether $recordDoesntExist()$ holds (line 22).

Any other access to $storedLabels_i[]$ merely updates cancelations or removes duplicates. Namely, canceling labels that are not the \preceq_{lb} -greatest among the ones that share the same creating processors (line 23) and canceling records that were canceled by other processors (line 24), as well as removing legitimate records that share the same ml (line 25). It is, therefore, clear that in any subsequent iteration of $receive()$ (after the first), $staleInfo()$ cannot hold. ■

Lemma 4.1 along with the lines 9 and 26 of the Algorithm, imply Corollary 4.1.

Corollary 4.1 *Consider a suffix R' of execution R that starts after the execution of a $receive()$ event. Then the following hold throughout R' : (i) $\forall p_i, p_j \in P$, the state of p_i encodes at most one legitimate label, $\ell_j =_{lCreator} j$ and (ii) ℓ_j can only appear in $storedLabels_i[j]$ and $max_i[]$ but not in $storedLabels_i[k] : k \neq j$.*

4.2.2 Local \preceq_{lb} -greatest legitimate local label

Lemma 4.2 considers processors for which $staleInfo()$ (line 9) does not hold. Note that $\neg staleInfo()$ holds at any time after the first step that includes the $receive()$ event (Lemma 4.1). Lemma 4.2 shows that p_i either adopts or creates the \preceq_{lb} -greatest legitimate local label and stores it in $max_i[i]$.

Lemma 4.2 *Let $p_i \in P$ be a processor such that $\neg staleInfo_i()$ (line 9), and $L_{pre}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j]) \wedge (\exists \langle max_i[j].ml, x \rangle \in (labels(max_i[j]) \setminus \{max_i[j]\}) \Rightarrow (x = \perp))\}$ be the set of $max_i[]$'s labels that, before p_i executes lines 21 to 28, are legitimate both in $max_i[]$ and in*

$storedLabels_i[]$'s queues. Let $L_{post}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ and $\langle \ell, \perp \rangle$ be the value of $max_i[i]$ immediately after p_i executes lines 21 to 28. The label $\langle \ell, \perp \rangle$ is the \preceq_{lb} -greatest legitimate label in $L_{post}(i)$. Moreover, suppose that $L_{pre}(i)$ has a \preceq_{lb} -greatest legitimate label, then that label is $\langle \ell, \perp \rangle$.

Proof. $\langle \ell, \perp \rangle$ is the \preceq_{lb} -greatest legitimate label in $L_{post}(i)$. Suppose that immediately before line 27, we have that $legitLabels_i() \neq \emptyset$, where $legitLabels_i() = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ (line 14). Note that in this case $L_{post}(i) = legitLabels_i()$. By the definition of \preceq_{lb} -greatest legitimate label and line 27, $max_i[i] = \langle \ell, \perp \rangle$ is the \preceq_{lb} -greatest legitimate label in $L_{post}(i)$. Suppose that $legitLabels_i() = \emptyset$ immediately before line 27, i.e., there are no legitimate labels in $\{max_i[j] : \exists p_j \in P\}$. By the definition of \preceq_{lb} -greatest legitimate label and line 15, $max_i[i] = \langle \ell, \perp \rangle$ is the \preceq_{lb} -greatest legitimate label in $L_{post}(i)$.

Suppose that $rec = \langle \ell', \perp \rangle$ is a \preceq_{lb} -greatest legitimate label in $L_{pre}(i)$, then $\ell = \ell'$. We show that the record rec is not modified in $max_i[]$ until the end of the execution of lines 21 to 28. Moreover, the records that are modified in $max_i[]$, are not included in $L_{pre}(i)$ (it is canceled in $storedLabels_i[]$) and no records in $max_i[]$ become legitimate. Therefore, rec is also the \preceq_{lb} -greatest legitimate label in $L_{post}(i)$, and thus, $\ell = \ell'$.

Since we assume that $staleInfo_i()$ does not hold, line 21 does not modify rec . Lines 22, 23 and 25 might add, modify, and respectively, remove $storedLabels_i$'s records, but it does not modify $max_i[]$. Since rec is not canceled in $storedLabels_i[]$ and the \preceq_{lb} -greatest legitimate label in $max_i[]$, the predicate $(legit(max[j]) \wedge notgeq(j))$ does not hold and line 23 does not modify rec . Moreover, the records in $max_i[]$, for which that predicate holds, become illegitimate. ■

4.2.3 Bounding the number of labels

Lemmas 4.3 and 4.4 present bounds on the number of adoption steps. These are $n + m$ for labels by labels that become inactive in any point in R and $(mn + 2n^2 - 2n)$ for any active processor. Following the above, choosing the queue sizes as $n + m$ for $storedLabels_i[j]$ if $i \neq j$, and $2(nm + 2n^2 - 2n) + 1$ for $storedLabels_i[i]$ is sufficient to prevent overflows given that m is the system's total link capacity in labels.

Maximum number of label adoptions in the absence of creations.

Suppose that there exists a processor, p_j , that has stopped adding labels to the system (the else part of line 28), say, because it became inactive (crashed), or it names a maximal label that is the \preceq_{lb} -greatest label among all the ones that the network ever delivers to p_j . Lemma 4.3 bounds the number of labels from p_j 's domain that any processor $p_i \in P$ adopts in R .

Lemma 4.3 *Let $p_i, p_j \in P$, be two processors. Suppose that p_j has stopped adding labels to the system configuration (the else part of line 28), and sending*

(line 16) these labels during R . Processor p_i adopts (line 27) at most $(n + m)$ labels, $\ell_j : (\ell_j =_{l_{Creator}} j)$, from p_j 's unknown domain ($\ell_j \notin \text{labels}_i(\ell_j)$) where m is the maximum number of label pairs that can be in transit in the system.

Proof. Let $p_k \in P$. At any time (after the first step in R) processor p_k 's state encodes at most one legitimate label, ℓ_j , for which $\ell_j =_{l_{Creator}} j$ (Corollary 4.1). Whenever p_i adopts a new label ℓ_j from p_j 's domain (line 27) such that $\ell_j : (\ell_j =_{l_{Creator}} j)$, this implies that ℓ_j is the only legitimate label pair in $\text{storedLabels}_i[j]$. Since ℓ_j was not transmitted by p_j before it was adopted, ℓ_j must come from p_k 's state delivered by a transmit event (line 16) or delivered via the network as part of the set of labels that existed in the initial arbitrary state. The bound holds since there are n processors, such as p_k , and m bounds the number of labels in transit. Moreover, no other processor can create label pairs from the domain of p_j . ■

Maximum number of label creations. Lemma 4.4 shows a bound on the number of adoption steps that does not depend on whether the labels are from the domain of an active or (eventually) inactive processor.

Lemma 4.4 *Let $p_i \in P$ and $L_i = \ell_{i_0}, \ell_{i_1}, \dots$ be the sequence of legitimate labels, $\ell_{i_k} =_{l_{Creator}} i$, from p_i 's domain, which p_i stores in $\text{max}_i[i]$ through the reception (line 17) or creation of labels (line 28), where $k \in \mathbb{N}$. It holds that $|L_i| \leq n(n^2 + m)$.*

Proof. Let $L_{i,j} = \ell_{i_0,j}, \ell_{i_1,j}, \dots$ be the sequence of legitimate labels that p_i stores in $\text{max}_i[j]$ during R and $C_{i,j} = \ell_{i_0,j}^c, \ell_{i_1,j}^c, \dots$ be the sequence of legitimate labels that p_i receives from processor p_j 's domain. We consider the following cases in which p_i stores L 's values in $\text{max}_i[i]$.

(1) **When $\ell_{i_k} = \ell_{j_0,j'}$, where $p_j, p_{j'} \in P$ and $k \in \mathbb{N}$.** This case considers the situation in which $\text{max}_i[i]$ stores a label that appeared in $\text{max}_j[j']$ at the (arbitrary) starting configuration, (i.e. $\ell_{j_0,j'} \in L_{j,j'}$). There are at most $n(n-1)$ such legitimate label values from p_i 's domain, namely $n-1$ arrays $\text{max}_j[]$ of size n .

(2) **When $\ell_{i_k} = \ell_{j_k,j'} = \ell_{j_0,j'}^c$, where $p_j, p_{j'} \in P$, $k, k' \in \mathbb{N}$ and $\ell_{j_k,j'} \neq \ell_{j_{k'},j'}$.** This case considers the situation in which $\text{max}_i[i]$ stores a label that appeared in the communication channel between p_j and $p_{j'}$ at the (arbitrary) starting configuration, (i.e. $\ell_{j_0,j'}^c \in C_{j,j'}$) and appeared in $\text{max}_j[j']$ before p_j communicated this to p_i . There are at most m such values, i.e., as many as the capacity of the communication links in labels, namely $|\mathcal{H}|$.

(3) **When ℓ_{i_k} is the return value of $\text{nextLabel}()$ (the else part of line 28).** Processor p_i aims at adopting the \preceq_{lb} -greatest legitimate label that is stored in $\text{max}_i[i]$, whenever such exists (line 27). Otherwise, p_i uses a label from its domain; either one that is the \preceq_{lb} -greatest legit label among the ones in $\text{storedLabels}_i[i]$, whenever such exists, or the returned value of $\text{nextLabel}()$ (line 28).

The latter case (the else part of line 28) refers to labels, ℓ_{i_k} , that p_i stores in $\text{max}_i[i]$ only after checking that there are no legitimate labels stored in $\text{max}_i[i]$

or $storedLabels_i[i]$. Note that every time p_i executes the else part of line 28, p_i stores the returned label, ℓ_{i_k} , in $storedLabels_i[i]$. After that, there are only three events for ℓ_{i_k} not to be stored as a legitimate label in $storedLabels_i[i]$: (i) execution of line 21, (ii) the network delivers to p_i a label, ℓ' , that either cancels ℓ_{i_k} or for which $\ell' \not\leq_{lb} \ell_{i_k}$, and (iii) ℓ_{i_k} overflows from $storedLabels_i[i]$ after exceeding the $(n(n^2 + m) + 1)$ limit which is the size of the queue.

Note that Lemma 4.1 says that event (i) can occur only once (during p_i 's first step). Moreover, only p_i can generate labels that are associated with its domain (in the else part of line 28). Each such label is \leq_{lb} -greater-equal than all the ones in $storedLabels_i[i]$ (by the definition of $nextLabel()$ in Algorithm 1).

Event (ii) cannot occur after p_i has learned all the labels $\ell \in remoteLabels_i$ for which $\ell \notin storedLabels_i[i]$, where $remoteLabels_i = (((\cup_{p_j \in P} localLabels_{i,j}) \cup \mathcal{H}) \setminus storedLabels_i[i])$ and $localLabels_{i,j} = \{\ell' : \ell' =_{lCreator} i, \exists p_j \in P : ((\ell' \in storedLabels_j[j]) \vee (\exists p_k \in P : \ell' = max_j[k].ml))\}$. During this learning process, p_i cancels or updates the cancellation labels in $storedLabels_i[i]$ before adding a new legitimate label. Thus, this learning process can be seen as moving labels from $remoteLabels_i$ to $storedLabels_i[i]$ and then keeping at most one legitimate label available in $storedLabels_i[i]$. Every time $storedLabels_i[i]$ accumulates a label ℓ that was unknown to p_i , the use of $nextLabel()$ allows it to create a label ℓ_{i_k} that is \leq_{lb} -greater than any label in $storedLabels_i[i]$ and eventually from all the ones in $remoteLabels_i$.

Note that $remoteLabels_i$'s labels must come from the (arbitrary) start of the system, because p_i is the only one that can add a label to the system from its domain and therefore this set cannot increase in size. These labels include those that are in transit in the system and all those that are unknown to p_i but exist in the $max_j[\bullet]$ or $storedLabels_j[j]$ structures of some other processor p_j . By Lemma 4.3 we know that $|storedLabels_j[j]| \leq n + m$ for $i \neq j$. From the three cases of L_i labels that we detailed at the beginning of this proof ((1)–(3)), we can bound the size of $remoteLabels_i$ as follows: for $p_j \in P : j \neq i$ we have that $|remoteLabels_i| = (n - 1)(|max[\bullet]| + |storedLabels_j[j]|) + |\mathcal{H}| = (n - 1)(n + (n + m)) + m = mn + 2n^2 - 2n$. Since p_i may respond to each of these labels with a call to $nextLabel()$, we require that $storedLabels_i[i]$ has size $2|remoteLabels_i| + 1$ label pairs in order to be able to accommodate all the labels from $|remoteLabels_i|$ and the ones created in response to these, plus the current greatest. Thus, what is suggested by event (ii) of p_i , i.e., receiving labels from $remoteLabels_i$, stops happening before overflows (event (iii)) occurs, since $storedLabels_i[i]$ has been chosen to have a size that can accommodate all the labels from $remoteLabels_i$ and those created by p_i as a response to these. This size is $2(mn + 2n^2 - 2n) + 1$ which is $O(n^3)$. ■

4.2.4 Pair diffusion

The proof continues and shows that active processors can eventually stop adopting or creating labels. We are particularly interested in looking into cases in which there are canceled label pairs and incomparable ones. We show that they eventually disappear from the system (Lemma 4.5) and thus no new labels are

Notation	Definition	Remark
$hName_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as <i>sentMax</i> feedback about $max_k[k]$
$hAck_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as <i>lastSent</i> feedback about $max_k[j]$
$max_{i,j,k}$	$\{max_j[j], max_k[k]\}$	Local maximal labels of p_j and p_k
$ack_{i,j,k}$	$\{max_j[j], max_k[j]\}$	ℓ_j is p_j 's local maximal label and $\ell_k = max_k[j]$
$stored_{i,j,k}$	$\{max_j[j]\} \times storedLabels_k[i]$	A label ℓ_k in $storedLabels_k[i]$ that can cancel $\ell_j = max_j[j]$

Table 1: The notation used to identify the possible positions of label pairs ℓ_j and ℓ_k that can cause canceling as used in Lemmas 4.5 to 4.7 and in Theorem 4.2.

being adopted or created (Lemma 4.6), which then implies the existence of a global maximal label (Lemma 4.7).

Lemmas 4.5 and 4.6, as well as Lemma 4.7 and Theorem 4.2 assume the existence of at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R . Suppose that processor $p_i \in P$ takes a bounded number of steps in R during a period in which $p_{unknown}$ takes a practically infinite number of steps. We say that p_i has become inactive (crashed) during that period and assume that it does not resume to take steps at any later stage of R (in the manner of fail-stop failures, as in Section 3).

Consider a processor $p_i \in P$ that takes any number of (bounded or practically infinite) steps in R and two processors $p_j, p_k \in P$ that take a practically infinite number of steps in R . Given that p_j has a label pair ℓ as its local maximal, and there exists another label pair ℓ' such that $\ell' \not\leq_{lb} \ell$ and they have the same creator p_i . Algorithm 2 suggests only two possible routes for some label pair ℓ' to find its way in the system through p_j . Either by p_j adopting ℓ' (line 27), or by creating it as a new label (the else part of line 28). Note, however, that p_j is not allowed to create a label in the name of p_i and since $\ell' =_{lCreator} i$, the only way for ℓ' to disturb the system is if this is adopted by p_j as in line 27. We use the following definitions for estimating whether there are such label pairs as ℓ and ℓ' in the system.

There is a *risk* for two label pairs from p_i 's domain, ℓ_j and ℓ_k , to cause such a disturbance when either they cancel one another or when it can be found that one is not greater than the other. Thus, we use the predicate $risk_{i,j,k}(\ell_j, \ell_k) = (\ell_j =_i \ell_k) \wedge legit(\ell_j) \wedge (notGreater(\ell_j, \ell_k) \vee canceled(\ell_j, \ell_k))$ to estimate whether p_j 's state encodes a label pair, $\ell_j =_{lCreator} i$, from p_i 's domain that may disturb the system due to another label, ℓ_k , from p_i 's domain that p_k 's state encodes, where $canceled(\ell_j, \ell_k) = (legit(\ell_j) \wedge \neg legit(\ell_k) \wedge \ell_j =_{ml} \ell_k)$ refers to a case in which label ℓ_j is canceled by label ℓ_k , $notGreater(\ell_j, \ell_k) = (legit(\ell_j) \wedge legit(\ell_k) \wedge \ell_k \not\leq_{lb} \ell_j)$ that refers to a case in which label ℓ_k is not \leq_{lb} -greater than ℓ_j and $(\ell_j =_i \ell_k) \equiv (\ell_j =_{lCreator} \ell_k =_{lCreator} i)$.

These two label pairs, ℓ_j and ℓ_k , can be the ones that processors p_j and p_k

name as their local maximal label, as in $max_{i,j,k} = \{(max_j[j], max_k[k])\}$, or recently received from one another, as in $ack_{i,j,k} = \{(max_j[j], max_k[j])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j , as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $stored_{i,j,k} = \{\{max_j[j]\} \times storedLabels_k[i]\}$. We define the union of these cases to be the set $risk = \{(\ell_j, \ell_k) \in max_{i,j,k} \cup ack_{i,j,k} \cup hName_{i,j,k} \cup hAck_{i,j,k} \cup stored_{i,j,k} : \exists p_i, p_j, p_k \in P \wedge stopped_j \wedge stopped_k \wedge risk_{i,j,k}(\ell_j, \ell_k)\}$, where $stopped_i = true$ when processor p_i is inactive (crashed) and $false$ otherwise. The above notation can also be found in Table 1.

Lemma 4.5 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R during a period where p_j never adopts labels (line 27), $\ell_j : (\ell_j =_{iCreator} i)$, from p_i 's unknown domain ($\ell_j \notin labels_j(\ell_j)$). Then eventually $risk = \emptyset$.*

Proof. Suppose this Lemma is false, i.e., the assumptions of this Lemmahold and yet in any configuration $c \in R$, it holds that $(\ell_j, \ell_k) \in risk \neq \emptyset$. We use $risk$'s definition to study the different cases. By the definition of $risk$, we can assume, without the loss of generality, that p_j and p_k are alive throughout R .

Claim: If p_j and p_k are alive throughout R , i.e. $stopped_j = stopped_k = False$, then $risk \neq \emptyset \iff risk_{i,j,k} = True$. This means that there exist two label pairs (ℓ_j, ℓ_k) where ℓ_k can force a cancellation to occur. Then the only way for this two labels to force $risk \neq \emptyset$ is if, throughout the execution, ℓ_k never reaches p_j .

The above claim is verified by a simple observation of the algorithm. If ℓ_k reaches p_j then lines 20, 24 and 26 guarantee a canceling and lines 22 and 23 ensure that these labels are kept canceled inside $storedLabels_j[]$. The latter is also ensured by the bounds on the labels given in Lemmas 4.3 and 4.4 that do not allow queue overflows. Thus to include these two labels to $risk$, is to keep ℓ_k hidden from p_j throughout R . We perform a case-by-case analysis to show that it is impossible for label ℓ_k to be “hidden” from p_j for an infinite number of steps in R .

The case of $(\ell_j, \ell_k) \in hName_{i,j,k}$. This is the case where $\ell_j = max_j[j]$ and ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $max_k[k]$. This may also contain such labels from the corrupt state. We note that p_j and p_k are alive throughout R . The stabilizing implementation of the data-link ensures that a message cannot reside in the communication channel during an infinite number of $transmit()$ – $receive()$ events of the two ends. Thus ℓ_k , which may well have only a single instance in the link coming from the initial corrupt state, will either eventually reach p_j or it become lost. In the both cases (the first by the Claim for the second trivially) the two clashing labels are removed from $risk$ and the result follows.

The case of $(\ell_j, \ell_k) \in hAck_{i,j,k}$. This is the case where $\ell_j = max_j[j]$ and

ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $\max_k[j]$. The proof line is exactly the same as the previous case.

This case follows by the same arguments to the case of $(\ell_j, \ell_k) \in \text{ack}_{i,j,k}$.

The case of $(\ell_j, \ell_k) \in \max_{i,j,k}$. Here the label pairs ℓ_j and ℓ_k are named by p_j and p_k as their local maximal label. We note that p_j and p_k are alive throughout R . By our self-stabilizing data-links and by the assumption on the communication that a message sent infinitely often is received infinitely often, then p_k transmits its $\max_k[k]$ label infinitely often when executing line 16. This implies that p_j receives ℓ_k infinitely often. By the Claim the canceling takes place, and the two labels are eventually removed from the global observer's *risk* set, giving a contradiction.

The case of $(\ell_j, \ell_k) \in \text{ack}_{i,j,k}$. This is the case where the labels (ℓ_j, ℓ_k) belong to $\{\max_j[j], \max_k[k]\}$. Since processor p_k continuously transmits its label pair in $\max_k[j]$ (line 16) the proof is almost identical to the previous case.

The case of $(\ell_j, \ell_k) \in \text{stored}_{i,j,k}$. This case's proof, follows by similar arguments to the case of $(\ell_j, \ell_k) \in \max_{i,j,k}$. Namely, p_k eventually receives the label pair $\ell_j = \max_j[j]$. The assumption that $\text{risk}_{i,j,k}(\ell_j, \ell_k)$ holds implies that one of the tests in lines 23 and 26 will either update $\text{storedLabels}_k[i]$, and respectively, $\max_k[j]$ with canceling values. We note that for the latter case we argue that p_j eventually received the canceled label pair in $\max_k[j]$, because we assume that p_j does not change the value of $\max_j[j]$ throughout R .

By careful and exhaustive examination of all the cases, we have proved that there is no way to keep ℓ_k hidden from p_j throughout R . This is a contradiction to our initial assumption, and thus eventually $\text{risk} = \emptyset$. ■

These two label pairs, ℓ_j and ℓ_k , can be the ones that processors p_j and p_k name as their local maximal label, as in $\max_{i,j,k} = \{(\max_j[j], \max_k[k])\}$, or recently received from one another, as in $\text{ack}_{i,j,k} = \{(\max_j[j], \max_k[k])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j , as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = \max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = \max_j[j] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $\text{stored}_{i,j,k} = \{\{\max_j[j]\} \times \text{storedLabels}_k[i]\}$.

Lemma 4.6 *Suppose that $\text{risk} = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{\text{unknown}} \in P$ whose identity is unknown, that takes practically infinite number of steps in R . Then p_j never adopts labels (line 27), $\ell_j : (\ell_j =_{\text{ICreator}} i)$, from p_i 's unknown domain ($\ell_j \notin \text{labels}_j(\ell_j)$).*

Proof. Note that the definition of *risk* considers almost every possible combination of two label pairs ℓ_j and ℓ_k from p_i 's domain that are stored by processor p_j , and respectively, p_k (or in the channels to them). The only combination that is not considered is $(\ell_j, \ell_k) \in \text{storedLabels}_j[i] \times \text{storedLabels}_k[i]$. However, this combination can indeed reside in the system during a legal execution and it cannot lead to a disruption for the case of $\text{risk} = \emptyset$ in every configuration throughout R because before that could happen, either p_j or p_k would have to adopt ℓ_j , and respectively, ℓ_k , which means a contradiction with the assumption that $\text{risk} = \emptyset$.

The only way that a label in $storedLabels[]$ can cause a change of the local maximum label and be communicated to also disrupt the system, is to find its way to $max[]$. Note that p_j cannot create a label under p_i 's domain (line 28) since the algorithm does not allow this, nor can it adopt a label from $storedLabels_j[i]$ (by the definition of $legitLabels()$, line 14). So there is no way for ℓ_j to be added to $max_j[j]$ and thus make $risk \neq \emptyset$ through creation or adoption.

On the other hand, we note that there is only one case where p_k extracts a label from $storedLabels_k[i] : i \neq k$ and adds it to $max_k[j]$. This is when it finds a legit label $\ell_j \in max_k[j]$ that can be canceled by some other label ℓ_k in $storedLabels_k[i]$, line 26. But this is the case of having the label pair (ℓ_j, ℓ_k) in $stored_{i,j,k}$. Our assumption that $risk = \emptyset$ implies that $stored_{i,j,k} = \emptyset$. This is a contradiction. Thus a label ℓ_k cannot reach $max_k[]$ in order for it to be communicated to p_j .

In the same way we can argue for the case of two messages in transit, $\mathcal{H}_{j,k} \times \mathcal{H}_{k,j}$ and that $risk = \emptyset$ throughout R . ■

Lemma 4.7 *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R . There is a legitimate label ℓ_{max} , such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((max_i[j] \preceq_{lb} \ell_{max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{max})))$.*

Proof. We initially note that the two processors p_i, p_j that take an infinite number of steps in R will exchange their local maximal label $max_i[i]$ and $max_j[j]$ an infinite number of times. By the assumption that $risk = \emptyset$, there are no two label pairs in the system that can cause canceling to each other that are unknown to p_i or p_j and are still part of $max_i[i]$ or $max_i[j]$. Hence, any differences in the local maximal label of the processors must be due to the labels' $lCreator$ difference.

Since $max_i[i]$ and $max_j[j]$ are continuously exchanged and received, assuming $max_i[i] \prec_{lb} max_j[j]$ where the labels are of different label creators, then p_i will be led to a $receive()$ event of $\langle sentMax_j, lastSent_j \rangle$ where $max_i[i] \prec_{lb} sentMax_j$. By line 19, $sentMax_j$ is added to $max_i[j]$ and since $risk = \emptyset$ no action from line 20 to line 26 takes place. Line 27 will then indicate that the greatest label in $max_i[\bullet]$ is that in $max_i[j]$ which is then adopted by p_i as $max_i[i]$, i.e., p_i 's local maximal. The above is true for every pair of processors taking an infinite number of steps in R and so we reach to the conclusion that eventually all such processors converge to the same ℓ_{max} label, i.e., it holds that $((max_i[j] \preceq_{lb} \ell_{max}) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{max})))$. ■

4.2.5 Convergence

Theorem 4.2 combines all the previous lemmas to demonstrate that when starting from an arbitrary starting configuration, the system eventually reaches a

configuration in which there is a global maximal label.

Theorem 4.2 *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R . Within a bounded number of steps, there is a legitimate label pair ℓ_{\max} , such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that p_i has $\max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((\max_i[j] \preceq_{lb} \ell_{\max}) \wedge (\forall \ell \in \text{storedLabels}_i[j] : \text{legit}(\ell)) \Rightarrow (\ell \preceq_{lb} \ell_{\max}))$.*

Proof. For any processor in the system, which may take any (bounded or practically infinite) number of steps in R , we know that there is a bounded number of label pairs, $L_i = \ell_{i_0}, \ell_{i_1}, \dots$, that processor $p_i \in P$ adds to the system configuration (the *else* part of line 28), where $\ell_{i_k} =_{lCreator} i$ (Lemma 4.4). Thus, by the pigeonhole principle we know that, within a bounded number of steps in R , there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst (all processors) p_i do not add any label pair, $\ell_{i_k} =_{lCreator} i$, to the system configuration (the *else* part of line 28).

During this practically infinite period (with respect to $p_{unknown}$), in which no label pairs are added to the system configuration due to the *else* part of line 28, we know that for any processor $p_j \in P$ that takes any number of (bounded or practically infinite) steps in R , and processor $p_k \in P$ that adopts labels in R (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j 's unknown domain ($\ell_j \notin \text{storedLabels}_k(j)$) it holds that p_k adopts such labels (line 27) only a bounded number times in R (Lemma 4.3). Therefore, we can again follow the pigeonhole principle and say that there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst neither p_i adds a label, $\ell_{i_k} =_{lCreator} i$, to the system (the *else* part of line 28), nor p_k adopts labels (line 27), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j 's unknown domain ($\ell_j \notin \text{labels}_k(\ell_j)$).

We deduce that, when the above is true, then we have reached a configuration in R where $risk = \emptyset$ (Lemma 4.5) and remains so throughout R (Lemma 4.6). Lemma 4.7 concludes by proving that, whilst $p_{unknown}$ takes a practically infinite number of steps, all processors (that take practically infinite number of steps in R) name the same \preceq_{lb} -greatest legitimate label which the theorem statement specifies. Thus no label $\ell =_{lCreator} j$ in $\max_i[\bullet]$ or in $\text{storedLabels}_i[j]$ may satisfy $\ell \not\preceq_{lb} \ell_{\max}$. ■

4.3 Increment Counter Algorithm

In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection to obtain a practically self-stabilizing counter increment algorithm.

Counters. To achieve this task, we now need to work with practically unbounded *counters*. As already mentioned in Section 2, a counter cnt is a triplet $\langle lbl, seqn, wid \rangle$, where lbl is an epoch label as defined in the previous subsection,

$seqn$ is a 64-bit integer sequence number and wid is the identifier of the processor that last incremented the counter's sequence number, i.e., wid is the counter writer. Then, given two counters cnt_i, cnt_j we define the relation $cnt_i \prec_{ct} cnt_j \equiv (cnt_i.lbl \prec_{lb} cnt_j.lbl) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn < cnt_j.seqn)) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn = cnt_j.seqn) \wedge (cnt_i.wid < cnt_j.wid))$. Observe that when the labels of the two counters are incomparable, the counters are also incomparable.

Therefore, the relation \prec_{ct} defines a total order (as required by practically unbounded counters) only when processors share a globally maximal label, (i.e., the system runs within a “stable” epoch). As we have shown in Theorem 4.2, starting from an arbitrary configuration, we eventually reach a configuration where the active processors have adopted the same maximal label. Essentially, the counter increment algorithm enhances the labeling algorithm to take care of the counter increment once such a maximal label exists in the system.

Enhancing the labeling algorithm to handle counters. Recall that in the labeling algorithm each processor p_i was maintaining two main structures of pairs of labels: array $max[]$ that stored the local maximal labels of each other processor (based on the message exchange) and $storedLabels[]$, an array of queues of label pairs that each processor maintains in an attempt to clean up obsolete labels created by itself or other processors. These structures now need to contain counters instead of just labels and are renamed to $maxC[]$ and $storedCnts[]$ (see line 1 of Algorithm 3). Each label can yield many different counters with different $\langle seqn, wid \rangle$. Therefore, in order to avoid increasing the size of these queues (with respect to the number of elements stored), we only keep the highest sequence number observed for each label (breaking ties with $wids$). We denote a counter pair by $\langle mct, cct \rangle$, with this being the extension of a label pair $\langle ml, cl \rangle$, where cct is a canceling counter for mct , such that either $cct.lbl \not\prec_{lb} mct.lbl$ (i.e., the counter is canceled), or $cct.lbl = \perp$.

Also, note that if there are counters in the system that are corrupt (being in the initial arbitrary configuration), then they can only force a change of label if their sequence number is *exhausted* (i.e., $seqn \geq 2^{64}$). Exhausted counters are treated by the counter algorithm in a way similar to the canceled labels in the labeling algorithm; an exhausted counter mct in a counter pair $\langle mct, cct \rangle$ is canceled, by setting $mct.lbl = cct.lbl$ (i.e., the counter's own label cancels it) and hence making the counter non-legit (thus it cannot be used as a local maximal counter in $maxC_i[i]$). This cannot increase the number of labels that are created due to the initially corrupted ones, as shown in the correctness proof that follows.

Another issue worth mentioning, is that the system is allowed to revert back to a previous legit label x , in case the current maximal label y becomes canceled. Label x might have been used before to create counters, so it is required to store the last sequence number written. If x is legit the system should not propose a new label and instead revert to x . Otherwise, the queues might grow with no bound. We enable reverting to such an x , by imposing

that each processor only stores a single instance of counters with the same label inside *storedCnts*[], namely the one with the maximal sequence number (*seqn*, *wid*). This is performed by storing the highest value of a counter that we hear about, as performed in line 19 upon a successful quorum write of a new sequence value, upon a receipt of any write request (line 31) and in every receipt of a counter through *receive()* by the definition of *process()*. Namely, in every possible appearance of a counter to the local state of a processor.

Quorums. We define a *quorum set* \mathbb{Q} based on processors in P , as a set of processor subsets of P (named *quorums*), that ensure a non-empty intersection of every pair of quorums. Namely, for all quorum pairs $Q_i, Q_j \in \mathbb{Q}$ such that $Q_i, Q_j \subset P$, it must hold that $Q_i \cap Q_j \neq \emptyset$. This *intersection property* is useful to propagate information among servers and exploiting the common intersection without having to write a value v to all the servers in a system, but only to a single quorum, say Q . If one wants to retrieve this value, then a call to *any* of the quorums (not necessarily Q), is expected to return v because there is least one processor in every quorum that also belongs to Q . In the counter algorithm we exploit the intersection property to retrieve the currently greatest counter in the system, increment it, and write it back to the system, i.e., to a quorum therein. Note that majorities form a special case of a quorum system.

Description of the Counter Algorithm. A pseudocode of the counter increment algorithm appears in Algorithm 3. The algorithm shows periodic counter operations (lines 12–14) –extending those of the labeling algorithm– and the counter increment operations (lines 15–31). The algorithm uses the enhanced counter structures *maxC*[n] and *storedCnts*[n] which are maintained in the same way as in the labeling algorithm with some additional operations. We define the operator *enqueue(ctp)* (line 3) to add a counter pair *ctp* to a queue of these structures if a corresponding counter with the same *lbl* doesn't exist, or to keep only one of the two instances if it exists. There are two enqueueing rules: (1) if at least one of the two counters is cancelled we keep a canceled instance, and (2) if both counters are legitimate we keep the greatest counter with respect to $\langle \textit{seqn}, \textit{wid} \rangle$. The counter is placed at the front of the queue.

Each processor p_i uses the token-based communication to transmit to every other processor p_j its own maximal counter and the one it currently holds for p_j in *maxC* _{i} [j] (line 12). Upon receipt of such an update from p_j , p_i first performs canceling of any exhausted counters in *storedCnts*[] (line 14), in *maxC*[] (line 14) and in the received couple of counter pairs (line 14). Having catered for exhaustion, it then calls *process*($\langle \bullet, \bullet \rangle$) with the received two counter pairs as arguments.

The *process()* operator calls lines 19 to 28 of Algorithm 2 adjusted for counter structures and handling counters. Thus, mentions to either labels or label structures in the labeling algorithm now refer to counters and counter structures. When adding to the counter queues the two enqueueing rules mentioned for *enqueue()* (above) hold. For ease of presentation we assume that

Algorithm 3: Counter Increment; code for p_i

1 **Variables:** A label lbl is extended to the triple $\langle lbl, seqn, wid \rangle$ called a *counter* where $seqn$, is the sequence number related to lbl , and wid is the identifier of the creator of this $seqn$. A counter pair $\langle mct, cct \rangle$ extends a label pair. cct is a canceling counter for mct , such that $cct.lbl \neq_{lb} mct.lbl$ or $cct.lbl = \perp$. We rename structures $max[]$ and $storedLabels[]$ of Alg. 2 to $maxC[]$ and $storedCnts[]$ that hold counter pairs instead of label pairs.

2 **Operators:** $process(\langle \bullet, \bullet \rangle)$ - executes the lines 19 to 28 of Algorithm 2 adjusted for counter structures and handling counters. For counter pairs with the same mct label, only the instance with the greatest counter w.r.t. \prec_{ct} is retained. In the case where one counter is cancelled we keep the cancelled. For ease of presentation we assume that a counter with a label created by p_i in line 28 of Algorithm 2, is initiated with a $seqn = 0$ and $wid = i$. A call of $process()$ (without arguments) essentially ignores lines 19 and 20 of Alg. 2.

3 $enqueue(ctp)$ - places a counter pair ctp at the front of a queue. If $ctp.mct.lbl$ already exists in the queue, it only maintains the instance with the greatest counter w.r.t. \prec_{ct} , placing it at the front of the queue. If one counter pair is canceled then the canceled copy is the one retained.

4 **Notation:** Let y and y' be two records that include the field x . We denote $y =_x y' \equiv (y.x = y'.x)$.

5 **Macros:**

6 $exhausted(ctp) = (ctp.mct.seqn \geq 2^{64})$

7 $legit(ctp) = (ctp.cct = \perp)$

8 $retCntrQ(ct) : \text{return } (storedCnts[ct.lbl.lCreator])$

9 $legitCounters() = \{maxC[j].mct : \exists p_j \in P \wedge legit(maxC[j])\}$

10 $cancelExhausted(ctp) : ctp.cct \leftarrow ctp.mct$

11 $cancelExhaustedMaxC() : \text{foreach } p_j \in P, c \in maxC[j] : exhausted(c) \text{ do}$
 $\quad cancelExhausted(maxC[j]) \text{ getMaxSeq() : return}$
 $\quad maxwid(\{max_{seqn}(\{ctp : ctp.mct \in legitCounters() \wedge maxC[i] =_{mct.lbl} ctp\}))$

// Lines 12 to 14 run in the background.

12 **upon** $transmitReady(p_j \in P \setminus \{p_i\})$ **do** $transmit(\langle maxC[i], maxC[j] \rangle);$

13 **upon** $receive(\langle sentMax, lastSent \rangle)$ **from** p_j **begin**

14 **foreach** $p_j \in P, ctp \in storedCnts[j] : legit(ctp) \wedge exhausted(ctp)$ **do**
 $cancelExhausted(ctp)$ **if** $(\exists ctp' \in \langle sentMax, lastSent \rangle : exhausted(ctp'))$ **then**
 $cancelExhausted(ctp') \text{ cancelExhaustedMaxC(); process}(\langle sentMax, lastSent \rangle);$

15 **procedure** $incrementCounter()$ **begin**

16 $quorumRead();$

17 **repeat** $findMaxCounter();$ **until** $legit(maxC[i]) \wedge \neg exhausted(maxC[i])$ **let**
 $newCntr = \langle maxC[i].mct.lbl, maxC[i].mct.seqn + 1, i \rangle;$

18 **if** $quorumWrite(newCntr)$ **then**

19 $maxC[i] \leftarrow newCntr; retCntrQ(maxC[i].mct).enqueue(maxC[i]);$

20 **procedure** $quorumRead()$ **begin**

21 **foreach** $p_j \in P$ **do** $send \text{ quorumMaxRead() while waiting for responses from a}$
 $quorum$ **do**

22 **upon** $receipt \text{ of } max^j \text{ from } p_j$ **do** $maxC[j] \leftarrow max^j;$

23 **upon** $request \text{ for quorumMaxRead() from } p_j$ **do** $\{findMaxCounter(); send \text{ maxC}_i[i]$
 to } p_j;

24 **procedure** $findMaxCounter()$ **begin**

25 $cancelExhaustedMaxC(); process();$

26 $maxC[i] \leftarrow getMaxSeq();$

27 **procedure** $quorumWrite(maxC_i[i])$ **begin**

28 **foreach** $p_j \in P$ **do** $send \text{ quorumMaxWrite(maxC}_i[i])$ **wait for ACK from a}**
 $quorum$

29 **upon** $request \text{ for quorumMaxWrite(max}^j \text{) from } p_j$ **begin**

30 $maxC_i[j] \leftarrow max_{ct}(max^j, maxC_i[j]);$

31 **if** $max_j =_{lbl.lCreator} i$ **then** $storedCnts_i[i].enqueue(maxC_i[i])$ **if**
 $exhausted(maxC_i[j])$ **then** $cancelExhausted(maxC_i[j])$ **send ACK to } p_j;**

a counter with a label created by p_i in line 28 of Algorithm 2, is initiated with a $seqn = 0$ and $wid = i$. A call to $process()$ (without arguments) essentially ignores lines 19 and 20 of Algorithm 2 and executes the rest of the lines performing bookkeeping tasks. After this call to $process()$, any exhausted counters from the initial arbitrary configuration, are enqueued as canceled to $storedCnts[]$. Therefore, they can never be readopted in case they are proposed with a non-exhausted counter.

The increment counter algorithm executed in lines 15 to 19 follows the logic of a writer in a MWMR register emulation. Processor p_i inquires the system for the counter they believe as greatest (line 16) by calling procedure $quorumRead()$ (lines 20–22). The responses contain the counter (max_j) that the responding processor p_j regards as the greatest (line 23). p_i aggregates the responses in its $maxC[]$ array. Note that there can be background counter diffusion as well. The $quorumRead()$ returns only when all the processors of one of the quorums have sent their responses (excluding responses from diffusion).

When the $quorumRead()$ completes, the $findMaxCounter()$ procedure is called repeatedly until a counter that is not canceled or exhausted is found; all counters that are exhausted must eventually become canceled. The function $findMaxCounter()$ cancels any exhausted counters in $maxC[]$ (while it holds the input from the quorum), and then calls $process()$ (line 25) to perform bookkeeping based on the new information and to provide a valid label. When the system is stabilized this label should not change. Any corrupt exhausted counter that might not have been canceled in the $storedCnts[]$ will, through the new call on $process()$, become canceled, making p_i immune from adopting it if it is proposed by other processors as valid. The $getMaxSeq()$ macro returns the maximal per \prec_{ct} , legit, non-exhausted counter it finds locally inside $maxC_i[]$. On exiting the loop (lines 17–17), the counter in $maxC_i[i]$ is the greatest of the counters returned by the quorum and any other processor (through diffusion), or, in case such a counter was not found, it is a newly created counter. As already stated such a counter is initiated to $seqn = 0$ and $wid = i$.

Following this, a local copy of $maxC_i[i]$ is incremented, i.e., the sequence number is increased by one, and wid is set to the identifier of p_i (line 17). The processor then attempts a write to the system (line 18) expecting responses from a quorum to return (line 27). Every processor p_j receiving p_i 's quorum write request, places it in $maxC_j[i]$ if it is greater than the value it already has in $maxC_j[j]$ and cancels it if it is exhausted. If the write fails for any reason to gather acknowledgments, the value does not get written to the local state as it does not satisfy the *if* condition of line 18.

Proof of correctness. We now prove the correctness of the counter algorithm. Initially we prove, that starting from an arbitrary configuration the system eventually reaches to a global maximal label (as given in Theorem 4.2), even in the presence of exhausted counters. We then continue to show that given such a global maximal label, the related counters are guaranteed to increment monotonically.

Lemma 4.3 *Consider two processors p_i taking a practically infinite number of steps and a setting as described by Theorem 4.2, adjusted for labels rather than counters as described above. Algorithm 3 guarantees that, within a bounded number of steps, every processor p_i holds a counter ct in $maxC_i[i]$ that has $ct.lbl = \ell_{max}$ the globally maximal label and ℓ_{max} is not exhausted. Moreover, ℓ_{max} is the greatest of all legitimate counter pair labels in $maxC_i[]$ and $storedCnts_i[]$.*

Proof. The proof follows the flow of the labeling algorithm proof, and provides minor amendments wherever the use of counters (instead of labels) challenges the correctness of the arguments. We show how the counter operations ensure that we reach to the globally maximal label ℓ_{max} becoming adopted by all the processors that take a practically infinite number of steps in execution R . We only require that $lbl = \ell_{max}$ while $seqn$ and wid may differ.

Key observation. Upon a receive event (lines 13–14) of the increment counter algorithm, lines 14, 14 and 14 cancel any exhausted counter pairs appearing as legitimate in $storedCnts[]$, $maxC[]$ and among the two received counter pairs by setting their mct as their cct . Increment counter procedures also have incoming counters. We note that any exhausted non-canceled counters stored in $maxC_i[]$ by a *quorumRead()*, are canceled by the immediate call of *cancelExhaustedMaxC()* in line 25 (through the call on *findMaxCntr()* of line 17). Incoming counters through *quorumWrite()* are also immediately checked for exhaustion on line 31.

In line with Lemma 4.1 we require that a full execution of a receive event has taken place, i.e., all lines 13 to 14 have been executed at least once. We now prove that all lemmas up to Lemma 4.4 in the labeling scheme’s correctness proof remain unaltered if we extend labels to counters and assume that the arbitrary state contains *exhausted* counters. The case of adopting an exhausted label which is then canceled, is an additional case in the body of the proof of Lemma 4.4 since all the other assumptions remain the same. Consider some processor $p_i \in P$ taking an infinite number of steps in execution R and assigning the label ℓ_x of an exhausted counter ct_x as $maxC_i[i]$. This implies that ℓ_x was not canceled when line 27 of Algorithm 2 was executed. By our key observation, any counter in the local state is checked for exhaustion and canceled immediately. By the assumption that at least one iteration of *receive* has taken place, we deduce that ℓ_x was adopted while canceled contradicting the conditions of line 27 of Algorithm 2 and the labeling algorithm proof. Thus, after a single iteration of *receive* it is impossible to adopt an exhausted label.

Exhausted counters cannot therefore increase adoptions and they pose no requirement for increasing the counter queue size, since we only keep a single instance of this canceled object. We note that once the canceling operations on exhausted counters take place, the call to *process* ensures that the canceled copies of these counters are retained in the $storedCnts[]$. Any new occurrences of these counter labels in $maxC[]$ are canceled by the corresponding canceled copies in $storedCnts[]$. From the arguments for label pair diffusion, which are identical for the counter pairs being diffused, any processor holding a counter ct_x as its

local maximal counter that is exhausted in the local state of some other active processor p_j , eventually stops using ct_x in favor of a counter with a different non-exhausted label. Following the results of the labeling algorithm, we deduce that our cancellation policy on the exhausted counters, enables Theorem 4.2 to also include the use of counters without any need to locally keep more counters than there are labels. By this theorem, we deduce that, eventually, any processor taking a practically infinite number of steps in R will have a counter with the globally maximal label ℓ_{max} . ■

Theorem 4.4 *Given an execution R of the counter increment algorithm in which at least a majority of processors take a practically infinite number of steps, the algorithm ensures that counters eventually increment monotonically.*

Proof. Given a suffix R' of the execution R in which Lemma 4.3 holds throughout, we define ct_{max} to be the counter with the globally maximal label that is the greatest in the system with respect to $\langle seqn, wid \rangle$. There are two cases:

Case 1: ct_{max} is the result of a call to the `incrementCounter()` procedure. Since this procedure only returned when `quorumWrite(ct_{max})` took place (line 28), therefore a quorum acknowledged the writing of this value. By the intersection property of the quorums, this counter was made known to at least one processor of every quorum. If there are concurrent writings of counters with the same *seqn* then the one with the greatest *wid* ensures monotonicity. Any subsequent call to `incrementCounter()` and thus to `quorumRead()` will, again by the intersection property of the quorums, return at least one instance of ct_{max} , since there is at least one processor in every quorum that acknowledged this counter.

Case 2: ct_{max} comes from the arbitrary state. By Lemmas 4.5, 4.6 and 4.7, the risk of having a label that remains hidden and that can cause a cancellation eventually becomes zero. We have previously used this proof to enforce that all exhausted counters eventually become canceled or are eliminated from the system. In the same vein we treat the case where ct_{max} is a remote counter that was not written to a quorum but may be revealed at some point to the system. Note that such a counter has the global maximal label and can indeed be adopted as a highest counter, since the adoption of this counter does not violate the monotonicity of counters, even if we go from one sequence number to a much greater one.

We also note, that this counter may have a sequence number near exhaustion. By the arguments of Case 1, the increments after this counter is adopted are monotonic and this will cause exhaustion of the counter requiring a label change in a number of increment steps that is not practically infinite. We have to mention here that this event does not increase the number of label creations, as the number of such counters that can cause eventual cancellation by exhaustion (after not practically infinite counter increments) is accounted for in the number of labels that can exist in the initial arbitrary state. The proof follows from our treatment of exhausted counters of Lemma 4.3.

Recall that our algorithm allows processor p_i to readopt a counter cnt_i with p_i 's own label that has a different label creator than the one it used in the

previous iteration of the labeling algorithm. Readoptions are only possible when cnt_i has not been canceled. In the case of such a readoption it is implied that cnt_i was dropped in favor of a counter cnt' with higher a $lCreator$ identifier that was eventually canceled. This implies that cnt' must come from the initial arbitrary configuration. Hence these “breaks” in monotonicity can only occur a bounded number of times in the execution, since counters such as cnt' are bounded in number and are handled by the Labeling algorithm.

Our algorithm stores every incoming counter with a label that was created by p_i in the $storedCnts_i[i]$ queue and by keeping the instance with the greatest $\langle seqn, wid \rangle$, (see lines 14, 19 and 31). So if p_i is to backstep to cnt_i , then the greatest instance that p_i has learned about cnt_i is adopted from $storedCnts_i[i]$. The only way for a new value of cnt_i to be missed by p_i , is for p_i to not hear of a quorum read incrementing p_i before cnt' was adopted. Again, as explained above, this is attributed to the bounded number of remnant counters from the arbitrary configuration that are dealt by the Labeling and Counter algorithms as Lemma 4.3 describes.

Now, under a legal execution where Lemma 4.3 holds, Case 2 can only occur a bounded number of times (since the counters in the initial arbitrary state are bounded in number). Furthermore, Case 1 is eventually true for the rest of the execution. In any case, the increment of the counter is monotonic with respect to \prec_{ct} in every subsequent call to $incrementCounter()$. ■

Having a self-stabilizing counter increment algorithm, it is not hard to implement a self-stabilizing MWMR register emulation. Each counter is associated with a value and the counter increment procedure essentially becomes a write operation: once the maximal counter is found, it is increased and associated with the new value to be written, which is then communicated to a majority of processors. The read operation is similar: a processor first queries all processors about the maximum counter they are aware of. It collects responses from a majority and if there is no maximal counter, it returns \perp so the processor needs to attempt to read again (i.e., the system hasn’t converged to a maximal label yet). If a maximal counter exists, it sends this together with the associated value to all the processors, and once it collects a majority of responses, it returns the counter with the associated value (the second phase is a standard requirement for preserving the consistency of the register (c.f. [3, 19])).

5 Virtually Synchronous Stabilizing Replicated State Machine

We now present a self-stabilizing reliable multicast algorithm that provides fault-tolerance, with respect to processor crashes and communication asynchrony, by considering the (*current group*) *view* of the changing processor set at the end of the group communication endpoint. We propose a self-stabilizing algorithm that guarantees the VS property. Namely, any two processors that are members of the same view, ought to deliver identical message sets to their SMRs as long as

they continue to share the same view, which indeed may change. This way, SMR algorithms can use the multicast service to synchronize their state transitions, i.e., the group members multicast their current automaton state, and the last received input that had led to that state.

Overview. A key advantage of multicast services (with virtual synchrony) is the ability to reuse the same view during many multicast rounds, and thus every automaton step requires a single multicast round. The aim of the proposed algorithm is to demonstrate in a self-stabilizing manner the most important ways to cut down the number of times in which the service needs to agree on a new view, and when it does, to perform it swiftly. Similar to [6], we assume that the service works in the network’s primary partition (see Definition 5.1) and require that a majority of processors are present in every view set. We do not however require all (local) failure detectors to agree on the set of recently alive and connected processors.

Multicast services that provide VS often leverage on the system’s ability to preserve (when possible) the coordinator during view transitions rather than electing a new coordinator. The motivation here is that the coordinator has the most recent automaton state and holds a copy of the set of *unstable* messages, which are the ones that were delivered to at least one view member, but the (alive and connected) view members have yet to receive a delivery acknowledgement for these. Our solution naturally follows this approach since it often helps the service to abstain from electing a leader upon every view change, as well as to avoid view transitions that require the coordinator to first investigate about all unstable messages (and the most recent automaton state) among all view members that continue to the next view. This is done so that the service can provide the virtual synchrony property. Thus, we consider the notion of coordinators that a majority of processors never suspects and we show that, in the existence of such processors, one of these coordinator will be eventually used in all subsequent views (Definition 5.1). As explained in Section 2, the algorithm, uses the counter increment algorithm, as well as a reliable multicast and a failure detector built over a self-stabilizing FIFO data link.

Definition 5.1 *We say that the output of the (local) failure detectors in execution R includes a primary partition when it includes a supporting majority of processors $P_{maj} : P_{maj} \subseteq P$, that (mutually) never suspect at least one processor, i.e., $\exists p_\ell \in P$ for which $|P_{maj}| > \lfloor n/2 \rfloor$ and $(p_i \in (P_{maj} \cap FD_\ell)) \iff (p_\ell \in (P_{maj} \cap FD_i))$ in every $c \in R$, where FD_x returns the set of processors that according to p_x ’s failure detector are active.*

5.1 Detailed Description of Algorithm 4

The existence of coordinator p_ℓ is in the heart of Algorithm 4. Processors that belong to and accept p_ℓ ’s view proposal are called the *followers* of p_ℓ . The algorithm determines the availability of a coordinator and acts towards the

Algorithm 4: A self-stabilizing automaton replication using virtual synchrony, code for processor p_i

```

1 Constants:  $PCE$  (periodic consistency enforcement) number of rounds between global
   state check;
2 Interfaces:  $fetch()$  next multicast message,  $apply(state, msg)$  applies the step  $msg$  to
    $state$  (while producing side effects),  $synchState(replica)$  returns a replica consolidated
   state,  $synchMsgs(replica)$  returns a consolidated array of last delivered messages,
    $failureDetector()$  returns a vector of processor pairs  $\langle pid, crdID \rangle$ ,  $inc()$  returns a counter
   from the increment counter algorithm;
3 Variables:  $rep[n] = \langle view = \langle ID, set \rangle, status \in \{Multicast, Propose, Install\}, (multicast$ 
   round number)  $rnd$ , (replica)  $state$ , (last delivered messages)  $msg[n]$  (to the state
   machine), (last fetched) input (to the state machine),  $propV = \langle ID, set \rangle$ , (no
   coordinator alive)  $noCrd$ , (recently live and connected component)  $FD$  : an array of
   state replica of the state machine, where  $rep[i]$  refers to the one that processor  $p_i$ 
   maintains. A local variable  $FDin$  stores the  $failureDetector()$  output.  $FD$  is an alias for
    $\{FDin.pid\}$ , i.e. the set of processors that the failure detector considers as active. Let
    $crd(j) = \{FDin.crdID : FDin.pid = j\}$ , i.e. the id of  $p_j$ 's local coordinator, or  $\perp$  if none.
4 Do forever begin
5   let  $FDin = failureDetector()$ ;
6   let  $seemCrd = \{p_\ell = rep[\ell].propV.ID.wid \in FD : (|rep[\ell].propV.set| > \lfloor n/2 \rfloor) \wedge$ 
    $(|rep[\ell].FD| > \lfloor n/2 \rfloor) \wedge (p_\ell \in rep[\ell].propV.set) \wedge (p_k \in rep[\ell].propV.set \leftrightarrow p_\ell \in$ 
    $rep[k].FD) \wedge ((rep[\ell].status = Multicast) \rightarrow (rep[\ell].view =$ 
    $propV) \wedge crd(\ell) = \ell)) \wedge ((rep[\ell].status = Install) \rightarrow crd(\ell) = \ell)\}$ ;
7   let  $valCrd = \{p_\ell \in seemCrd : (\forall p_k \in seemCrd : rep[k].propV.ID \preceq_{ct}$ 
    $rep[\ell].propV.ID)\}$ ;
8    $noCrd \leftarrow (|valCrd| \neq 1); crdID \leftarrow valCrd;$ 
9   if  $(|FD| > \lfloor n/2 \rfloor) \wedge ((|valCrd| \neq 1) \wedge (\{p_k \in FD : p_i \in rep[k].FD \wedge$ 
    $rep[k].noCrd\} > \lfloor n/2 \rfloor)) \vee ((valCrd = \{p_i\}) \wedge (FD \neq propV.set) \wedge (\{p_k \in FD :$ 
    $rep[k].propV = propV\} > \lfloor n/2 \rfloor)))$  then  $(status, propV) \leftarrow (Propose, \langle inc(), FD \rangle)$ 
10  else if  $(valCrd = \{p_i\}) \wedge (\forall p_j \in view.set : rep[j].(view, status, rnd) = (view,$ 
    $status, rnd)) \vee ((status \neq Multicast) \wedge (\forall p_j \in propV.set :$ 
    $rep[j].(propV, status) = (propV, Propose)))$  then
11    if  $status = Multicast$  then
12       $apply(state, msg); input \leftarrow fetch();$ 
13      foreach  $p_j \in P$  do if  $p_j \in view.set$  then  $msg[j] \leftarrow rep[j].input$  else
    $msg[j] \leftarrow \perp$   $rnd \leftarrow rnd + 1;$ 
14    else if  $status = Propose$  then
    $(state, status, msg) \leftarrow (synchState(rep), Install, synchMsgs(rep))$  else if
    $status = Install$  then  $(view, status, rnd) \leftarrow (propV, Multicast, 0)$ 
15  else if
    $valCrd = \{p_\ell\} \wedge \ell \neq i \wedge ((rep[\ell].rnd = 0 \vee rnd < rep[\ell].rnd \vee rep[\ell].(view \neq propV))$ 
   then
16    if  $rep[\ell].status = Multicast$  then
17      if  $rep[\ell].state = \perp$  then  $rep[\ell].state \leftarrow state$  /* PCE optimization, line 21 */
    $rep[i] \leftarrow rep[\ell]; apply(state, rep[\ell].msg);$  /* for the sake of side-effects */
18       $input \leftarrow fetch();$ 
19    else if  $rep[\ell].status = Install$  then  $rep[i] \leftarrow rep[\ell]$  else if  $rep[\ell].status = Propose$ 
   then  $(status, propV) \leftarrow rep[\ell].(status, propV)$ 
20  let  $m = rep[i]$  /* sending messages: all to coordinator and coordinator to all */ ;
21  if  $status = Multicast \wedge rnd \pmod{PCE} \neq 0$  then  $m.state \leftarrow \perp$  /* PCE optimization,
   line 17 */
22  let  $sendSet = (seemCrd \cup \{p_k \in propV.set : valCrd = \{p_i\}\} \cup \{p_k \in FD : noCrd \vee$ 
    $(status = Propose)\})$ 
23  foreach  $p_j \in sendSet$  do  $send(m)$ 
24 Upon message arrival  $m$  from  $p_j$  do  $rep[j] \leftarrow m;$ 

```

election of a new one when no valid such exists (lines 5 to 9). The pseudocode details the coordinator-side (lines 10 to 14) and the follower-side (lines 15 to 19) actions. At the end of each iteration the algorithm defines how p_ℓ and its followers exchange messages (lines 21 to 24).

The processor state and interfaces. The state of each processor includes its current *view*, and *status* = {Propose, Install, Multicast}, which refers to usual message multicast operation when in Multicast, or view establishment rounds in which the coordinator can Propose a new view and proceed to Install it once all preparations are done (line 3). During multicast rounds, *rnd* denotes the round number, *state* stores the replica, *msg*[*n*] is an array that includes the last delivered messages to the state machine, which is the *input* fetched by each group member and then aggregated by the coordinator during the previous multicast round. During multicast rounds, it holds that $propV = view$. However, whenever $propV \neq view$ we consider *propV* as the newly proposed view and *view* as the last installed one. Each processor also uses *noCrd* and *FD* to indicate whether it is aware of the absence of a recently active and connected valid coordinator, and respectively, of the set of processor present in the connected component, as indicated by its local failure detector. The processors exchange their state via message passing and store the arriving messages in the replica's array, *rep*[*n*] (line 24), where *rep*[*i*].(*view*, ..., *noCrd*) is an alias to the aforementioned variables and *rep*[*j*] refers to the last arriving message from processor p_j containing p_j 's *rep*[*j*]. Our presentation also uses subscript k to refer to the content of a variable at processor p_k , e.g., *rep* _{k} [*j*].*view*, when referring to the last installed view that processor p_k last received from p_j .

Algorithm 4 assumes access to the application's message queue via *fetch*(), which returns the next multicast message, or \perp when no such message is available (line 2). It also assumes the availability of the automaton state transition function, *apply*(*state*, *msg*), which applies the aggregated input array, *msg*, to the replica's *state* and produces the local side effects. The algorithm also collects the followers' replica states and uses *synchState*(*replica*) to return the new state. The function *failureDetector*() provides access to p_i 's failure detector, and the function *inc*() (counter increment) fetches a new and unique (view) identifier, *ID*, that can be totally ordered by \preceq_{ct} and *ID.wid* is the identity of the processor that incremented the counter, resulting to the counter value *ID* (hence view *ID*s are counters as defined in Section 4.3). Note that when two processors attempt to concurrently increment the counter, due to symmetry breaking, one of the two counters is the largest. Each processor will continue to propose a new view based on the counter written, but then (as described below) the one with the highest counter will succeed (line 7).

Determining coordinator availability. Algorithm 4 takes an agile approach to message multicasting with atomic delivery guarantees. Namely, a new view is installed whenever the coordinator sees a change to its local failure detector, *failureDetector*(), which p_i stores in *FD* _{i} (line 5). Processor p_i can see

the set of processors, $seemCrd_i$, that each “seems” to be the view coordinator, because p_i stored a message from $p_\ell \in FD_i$ for which $p_\ell = rep[\ell].propV.ID.wid$. Note that p_i cannot consider p_ℓ as a (seemly) coordinator when p_ℓ ’s proposal view does not include a majority, or if p_ℓ is not a member in the view it claims to coordinate. In the case of **Multicast** rounds, their view fields must match their view proposal fields (line 6). Also, using the failure detector heartbeat exchange, processors communicate the identifier of the processor they consider to be their coordinator, or \perp if none. As shown in the correctness proof, this helps to detect initially corrupted states where a processor p_i might consider processor p_j to be its coordinator, but processor p_j does not consider itself to be the coordinator.

The algorithm considers a processor as the valid coordinator, if it belongs to $seemCrd$ and has the \preceq_{ct} -greatest view identifier among the set of seemly coordinators (line 7). Note that the set $valCrd_i$ either includes a single processor, p_ℓ which p_i considers to be a valid coordinator, or p_i does not consider any processor to be a valid coordinator that was recently live and connected (line 8). In the latter case, p_i will not propose a new view before its (local) failure detector indicates that it is within the primary component and that a supportive majority of recently live and connected processors also do not observe the availability of a valid coordinator (line 9). Note that in the case where p_i is a valid coordinator, it will create and propose a new view whenever the last proposed view does not match the set of processors that were recently live and connected according to its (local) failure detector. In such a case no other processor but p_i may propose, because it is the only one that retains a majority of processors that have accepted the previous view.

The coordinator-side. Processor p_i is aware of its valid coordinatorship when $(valCrd_i = \{p_i\})$ (line 10). It takes action related to its role as a coordinator when it detects the round end, based on input from other processors. During a normal **Multicast** round, p_i observes the round end once for every view member p_j it holds that $(rep_i[j].(view, status, rnd) = (view_i, status_i, rnd_i))$. For the case of **Propose** and **Install** rounds, the algorithm does not need to consider the round number, rnd .

Depending on its *status*, the coordinator p_i proceeds once it observes the successful round conclusion. At the end of a normal **Multicast** round, the coordinator increments the round number after aggregating the followers’ input (line 11). The coordinator continues from the end of a **Propose** round to an **Install** round after using the most recently received replicas to install a synchronized state of the emulated automaton (line 14). At the end of a successful **Install** round, the coordinator proceeds to a **Multicast** round after installing the proposed view and the first round number. (Note that implicitly the coordinator creates a new view if it detects that the round number is exhausted ($rnd > 2^{64}$), or if there is another member of its view that has a greater round number than the one this coordinator has. This can only be due to corruption in the initial arbitrary state which affected rnd part of the state.)

The follower-side. Processor p_i is aware of its coordinator's identity when ($valCrd_i = \{p_\ell\}$) and $i \neq \ell$ (line 15). Being a follower, p_i only enters this block of the pseudocode when it receives a new message, i.e., the first message round when installing a new view ($rep[\ell].rnd = 0$), the first time a message arrives ($rnd < rep[\ell].rnd$) or a new view is proposed ($rep[\ell].view \neq propV$).

During normal Multicast rounds (line 16) the follower p_i applies the aggregated message of this round to its current automaton state so that it produces the needed side-effects before adopting the coordinator's replica (line 19). Note that, in the case of a Propose round, the algorithm design stops p_i from overwriting its round number, thus allowing the coordinator to know what was the last round number that it delivered during the last installed view.

The exchanging message and PCE optimization. Each processor periodically sends its current replica (line 23) and stores the received ones (line 24). As an optimization, we propose to avoid sending the entire replica state in every Multicast round. Instead, we consider a predefined constant, *PCE* (periodic consistency enforcement), that determines the maximum number of Multicast rounds during which the followers do not transmit their replica state to the coordinator and the coordinator does not send its state to them (lines 17 and 21). Note that the greater the *PCE*'s size, the longer it takes to recover from transient faults. Therefore, one has to take this into consideration when extending the approach of periodic consistency enforcement to other elements of replica, e.g., in *view* and *propV*, one might want to reduce the communication costs that are associated with the *set* field and the epoch part of the *ID* field.

5.2 Correctness Proof of Algorithm 4

The correctness proof shows that starting from an arbitrary state in an execution R of Algorithm 4 and once the primary partition property (Definition 5.1) holds throughout R , we reach a configuration $c \in R$ in which some processor with supporting majority p_ℓ will propose a view including its supporting majority. This view is either accepted by all its member processors or in the case where p_ℓ experiences a failure detection change, it can repropose a view. We conclude by proving that any execution suffix of R that begins from such a configuration c will preserve the virtual synchrony property and implement state machine replication. We begin with some definitions.

Once the system considers processor p_ℓ as the view coordinator (Definition 5.1) its supporting majority can extend the support throughout R and thus p_ℓ continues to emulate the automaton with them. Furthermore, there is no clear guarantee for a view coordinator to continue to coordinate for an unbounded period when it does not meet the criteria of Definition 5.1 throughout R . Therefore, for the sake of presentation simplicity, the proof considers any execution R with only *definitive suspicions*, i.e., once processor p_i suspects processor p_j , it does not stop suspecting p_j throughout R . The correctness proof implies that eventually, once all of R 's suspicions appear in the respective local

failure detectors, the system elects a coordinator that has a supporting majority throughout R .

Consider a configuration c in an execution R of Algorithm 4 and a processor $p_i \in P$. We define the *local (view) coordinator* of p_i , say p_j , to be the only processor that, based on p_i 's local information, has a proposed view satisfying the conditions of lines 6 and 7 such that $valCrd = \{p_j\}$. p_j is also considered the *global (view) coordinator* if for all p_k in p_j 's proposed view ($propV_j$), it holds that $valCrd_k = \{p_j\}$. When p_i has a (local) coordinator then p_i 's local variable $noCrd = \text{False}$, whilst when it has no local coordinator, $noCrd = \text{True}$. Moving to the proof, we consider the following useful remark on Definition 5.1 of page 31.

Remark 5.1 *Definition 5.1 suggests that we can have more than one processor that has supporting majority. In this case, it is not necessary to have the same supporting majority for all such processors. Thus for two such processors p_i, p_j with respective supporting majorities $P_{maj}(i)$ and $P_{maj}(j)$ we do not require that $P_{maj}(i) = P_{maj}(j)$, but $P_{maj}(i) \cap P_{maj}(j) \neq \emptyset$ trivially holds.*

Lemma 5.1 *Let R be an execution with an arbitrary initial configuration, of Algorithm 4 such that Definition 5.1 holds. Consider a processor $p_i \in P_{maj}$ which has a local coordinator p_k , such that p_k is either inactive or it does not have a supporting majority throughout R . There is a configuration $c \in R$, after which p_i does not consider p_k to be its local coordinator.*

Proof. There are the two possibilities regarding processor p_k .

Case 1: We first consider the case where p_k is inactive throughout R . By the design of our failure detector, p_i is informed of p_k 's inactivity such that line 5 will return an FD_i to p_i where $p_k \notin FD_i$. The threshold we set for our failure detector (see Section 2) determines how soon p_k is suspected. By the first condition of line 6 we have that $p_k \notin FD_i \Rightarrow p_k \notin seemCrd \Rightarrow p_k \notin valCrd_i$, i.e., p_i stops considering p_k as its local coordinator. By definitive suspicions, that p_i does not stop suspecting p_k throughout R .

We now turn to the case where p_k is active, however it does not have a supporting majority throughout R , but p_i still considers p_k as its local coordinator, i.e. $valCrd_i = \{p_k\}$. Two subcases exist:

Case 2(a): p_k considers itself to have a supporting majority, and $p_i \in propV_k$. Note that the latter assumption implies that p_k is forced by lines 20 - 23 to propagate $rep_k[k]$ to p_i in every iteration. By the failure detector, there exists an iteration where p_k will have $|FD_k| = n/2 + 1$ and is informed that some $p_j \in propV_k$ has $p_k \notin FD_j$ and so the condition of line 6 ($FD > \lfloor n/2 \rfloor$) fails for p_k , which stops being the coordinator of itself. If p_k does not find a new coordinator, hence $noCrd_k = \text{True}$, then p_k propagates its $rep_k[k]$ to p_i . But this implies that p_i receives $rep_k[k]$ and stores it in $rep_i[k]$. Upon the next iteration of this reception, p_i will remove p_k from its $seemCrd$ set because p_k does not satisfy the condition $|rep_i[k].FD| < \lfloor n/2 \rfloor$ of line 6. We conclude that p_i stops considering p_k as its local coordinator if p_k does not find a new

coordinator. Nevertheless, p_k may find a new coordinator before propagating $rep_k[k]$. If p_k has a coordinator other than itself, then it only propagates $rep_k[k]$ to its coordinator and thus p_i does not receive this information. We thus refer to the next case:

Case 2(b): p_k has a different local coordinator than itself. This can occur either as described in Case 2(a) or as a result of an arbitrary initial state in which p_i believes that p_k is its local coordinator but p_k has a different local coordinator. We note that the difficulty of this case is that p_k only sends $rep_k[k]$ to its coordinator, and thus the proof of Case 2(a) is not useful here. As explained in Algorithm 4, the failure detector returns a set with the identities (pid) of all the processors it regards as active, as well as the identity of the local coordinator of each of these processors. As per the algorithm's notation, the coordinator of processor p_k is given by $crd(k)$. Since p_i 's failure detector regards p_k as active, then $crd(k)$ is indeed updated (remember that p_i receives the token with p_k 's $crd(k)$ infinitely often from p_k), otherwise p_k is removed from FD and is not a valid coordinator for p_i . But p_k does not consider itself as the coordinator (by the assumption of Case 2(b)), and thus it holds that $crd(k) \neq k$. Therefore, in the first iteration after p_i receives $crd(k) \neq k$, one of the last two conditions of line 6 fails (depending on what is the view status that p_i has in $rep_i[k]$) so $p_k \notin seemCrd_i$ and thus $valCrd_i \neq \{p_k\}$. We conclude that any such p_k stops being p_i 's coordinator and by the assumption of definitive suspicions we reach to the result. It is also important to note that p_k never again satisfies all the conditions of line 9 to create a new view. ■

We now define the notion of “propose” more rigorously to be used in the sequel.

Definition 5.2 *Processor $p_\ell \in P$ with status = Propose, is said to propose a view $propV_\ell$, if in a complete iteration of Algorithm 4, p_ℓ either satisfies $valCrd_\ell = \{p_\ell\}$ or satisfies all the conditions of line 9 to create $propV_\ell$. A proposal is completed when $propV_\ell$ is propagated through lines 20–23 to all the members of FD_ℓ .*

The above definition does not imply that p_ℓ will continue proposing the view $propV$, since the replicas received from other processors may force p_ℓ to either exclude itself from $valCrd_\ell$ or create a new view (see Lemma 5.3). If the view is installed, then the proposal procedure will stop, although $propV_\ell$ will still be sent as part of the replica propagation at the end of each iteration. Also note that the origins of such a proposed view are not defined. Indeed it is possible for a view that was not created by p_ℓ but bears p_ℓ 's creator identity to come from an arbitrary state and be proposed, as long as all the conditions of lines 6 and 7 are met.

Lemma 5.2 *If the conditions of Definition 5.1 hold throughout an execution R of Algorithm 4, then starting from an arbitrary configuration in which there is no global coordinator, the system reaches a configuration in which at least one processor with a supporting majority will propose a view (with “propose” defined as in Definition 5.2).*

Proof. By Definition 5.1, at least one processor with supporting majority exists. Denote one such processor as p_ℓ . Assume for contradiction that throughout R , no processor p_ℓ with supporting majority proposes a view. p_ℓ either has a local coordinator (that is not global) or does not have a coordinator.

Case 1: p_ℓ does not have a coordinator ($\text{noCrd}_\ell = \text{True}$). If p_ℓ does not propose a view (as per the “propose” Definition 5.2), this is because it does not hold a proposal that is suitable and it does not satisfy some condition of line 9 which would allow it to create a new view. The first condition of line 9, $(|FD| > \lfloor n/2 \rfloor)$ is always satisfied by our assumption that p_ℓ is not suspected by a majority throughout R . In the second condition, both (i) $((|\text{valCrd}_\ell| \neq 1) \wedge (|\{p_i \in FD_\ell : p_i \in \text{rep}_\ell[i].FD_\ell \wedge \text{rep}_\ell[i].\text{noCrd}\}| > \lfloor n/2 \rfloor))$ and (ii) $((\text{valCrd}_\ell = \{p_\ell\}) \wedge (FD_\ell \neq \text{propV}_\ell.\text{set}) \wedge (|\{p_i \in FD : \text{rep}[i].\text{propV} = \text{propV}\}| > \lfloor n/2 \rfloor))$ must fail due to our assumption that p_ℓ never proposes. Indeed (ii) fails since $\text{noCrd}_\ell = \text{True} \Rightarrow \text{valCrd}_\ell \neq \{p_\ell\}$. If the first expression also fails, this implies that throughout R , p_ℓ does not know of a majority of processors with $\text{noCrd} = \text{True}$ and so it cannot propose a new view.

Let’s assume that only one processor $p_j \in P_{\text{maj}}(\ell) \subseteq FD_\ell$ is required to switch from $\text{noCrd}_j = \text{False}$ to True in order for p_ℓ to gain a majority of processors without a coordinator. But if $\text{noCrd}_j = \text{False}$ then p_j must already have a coordinator, say p_k . We have the following two subcases:

Case 1(a): p_k does not have a supporting majority. Lemma 5.1 guarantees that p_j stops considering p_k as its local coordinator. Thus p_j eventually goes to $\text{noCrd} = \text{True}$ and by the propagation of its replica, p_ℓ receives the required majority to go into proposing a view. But this contradicts our initial assumption, so we are lead to the following case.

Case 1(b): p_k has a supporting majority and a view proposal propV_k from the initial arbitrary configuration but is not the global coordinator. But this implies that the Lemma trivially holds, and so the following case must be true.

Case 2: p_ℓ has a coordinator, say $p_{k'}$. The two subcases of whether $p_{k'}$ has a supporting majority or not, are identical to the two subcases 1(a) and 1(b) concerning p_k that we studied above. Thus, it must be that either p_ℓ will eventually propose a label, or that $p_{k'}$ has a proposed view, thus contradicting our assumption and so the lemma follows. ■

Lemma 5.2 establishes that at least one processor with supporting majority will propose a view in the absence of a valid coordinator. We now move to prove that such a processor will only propose one view, unless it experiences changes in its FD that render the view proposal’s membership obsolete. The lemma also proves that any two processors with supporting majority will not create views in order to compete for the coordinatorship.

Lemma 5.3 *If the conditions of Definition 5.1 hold throughout an execution R of Algorithm 4, then starting from an arbitrary configuration, the system reaches a configuration in which any processor p_ℓ with a supporting majority proposes a view propV_ℓ , and cannot create a new proposed view in R unless $FD_\ell \neq \text{propV}_\ell.\text{set}$ and a majority of processors has adopted propV_ℓ . As a consequence, the system reaches a configuration in which one processor with*

supporting majority is the global coordinator until the end of the execution.

Proof. We distinguish the following cases:

Case 1: Only one processor with supporting majority exists. Assume there is only a single processor p_ℓ that has a supporting majority throughout R . According to Lemma 5.2, p_ℓ must eventually propose a view $propV_\ell$, based on the current FD_ℓ reading (line 5) which becomes the $propV_\ell.set$. By Lemma 5.1, any other processor without a supporting majority will eventually stop being the local coordinator of any $p_j \in propV_\ell.set$ and since such processors do not have a supporting majority, the first condition of line 9 will prevent them from proposing.

Processor p_ℓ continuously proposes $propV_\ell$ until all processors in $propV_\ell.set$ have sent a replica showing that they have adopted $propV_\ell$ as their $propV$. Every processor that is alive throughout R and in FD_ℓ should receive this replica through the self-stabilizing reliable communication. The only condition that may prevent p_j to adopt $propV_\ell$ is if for some $p_r \in rep_j[\ell].propV_\ell.set$ it holds that $p_\ell \notin rep_j[r].FD$ (line 6). Plainly put, p_j believes that p_r suspects p_ℓ .

Case 1(a): If p_j 's information is correct about p_r , then $p_r \notin P_{maj}(\ell)$. Thus at some point p_ℓ will suspect p_r and exclude p_r FD_ℓ .

Case 1(b): If p_j 's information is false –remnant of some arbitrary state–, then $p_\ell \in FD_r$ and since p_r , by the last condition of line 22, sends $rep_r[r]$ infinitely often to p_j , then $rep_j[r]$ will be corrected and p_j will accept $propV_\ell$.

Since p_ℓ has a majority $P_{maj}(\ell) \subseteq propV_\ell.set$, then at least a majority of processors have received $propV_\ell$ and eventually accept it. If some processor $p_j \in propV_\ell$ does not adopt p_ℓ 's proposal in R , it is eventually removed from FD_ℓ and thus does not belong to the supporting majority of p_ℓ (as detailed in Case 1(a) above). By the above we note that p_ℓ is able to get at least the supporting majority $P_{maj}(\ell)$ to accept its view if not all of the members in $propV_\ell.set$. In the last case it can proceed to the installation of the view. If there is any change in the failure detector of p_ℓ before it installs a view, p_ℓ can satisfy the second case of line 9, to create a new updated view. Note that in the mean time no processor other than p_ℓ can satisfy the conditions of that line, and thus it is the only processor that can propose and become the coordinator. Thus p_ℓ eventually becomes the coordinator if it is the single majority-supported processor.

Case 2: More than one processor with supporting majority. Consider two processors $p_\ell, p_{\ell'}$ that have a supporting majority such that each creates a view (line 9). By the correctness of our counter algorithm, $inc()$ returns two distinct and ordered counters to use as view identifiers. Without loss of generality, we assume that $propV_\ell$ proposed by p_ℓ has the greatest identifier of all the counters created by calls to $inc()$. We identify the following four subcases:

Case 2(a): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. In this case $p_{\ell'}$ will propose its view $propV_{\ell'}$ and wait for all $p_i \in propV_{\ell'}.set$ to adopt it (line 10). Whenever p_ℓ receives $propV_{\ell'}$, it will store it but will not adopt it, since $propV_{\ell'}.ID \preceq_{ct} propV_\ell.ID$ (line 7). The proposal $propV_{\ell'}$ is also propagated to every $p_i \in$

$propV_\ell.set$. Since there is no greater proposed view identifier than $propV_\ell.ID$, this is adopted by all $p_i \in propV_\ell$ which also includes $p_{\ell'}$ as well. Thus any processor with supporting majority that belonged to the proposed set of p_ℓ will propose at most once, and p_ℓ will become the sole coordinator. Note that if $p_{\ell'}$ is prevented from adopting $propV_\ell$ for some time, this is due to reasons detailed and solved in Case 1 of the previous lemma. The case where the failure detection reading changes for p_ℓ is also tackled as in Case 1 of this lemma, by noticing that if p_ℓ manages to get a majority of processors of $propV.set$ then p_ℓ will change its proposed view without losing this majority.

Case 2(b): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Since both processors were able to propose, this implies that a majority of processors that belonged to each of p_ℓ 's and $p_{\ell'}$'s supporting majority had informed that they had no coordinator (line 9). Each of p_ℓ and $p_{\ell'}$, proposes its view to its $propV.set$, and waits for acknowledgments from *all* the processors in $propV.set$ (line 10), in order to install the view. Since $p_\ell \notin FD_{\ell'}$, $p_{\ell'}$ does not consider $propV_\ell$ a valid proposal (line 6) and retains its own proposal that it propagates. The same is done by p_ℓ . Since p_ℓ has the greatest label, any $p_i \in propV_\ell.set \cap propV_{\ell'}.set$ might initially adopt $propV_{\ell'}$ but it will eventually choose the greatest $propV_\ell$. If $p_{\ell'}$'s proposal was accepted by all members of $propV_{\ell'}$ then this means that $p_{\ell'}$ became the global coordinator but will then lose the coordinatorship to p_ℓ because $propV_\ell$ has a greater view identifier.

What is more crucial, is that $p_{\ell'}$ cannot make another proposal, since it will not have a majority of processors that do not have a coordinator. This is deduced from the intersection property of the two majorities ($propV_\ell.set$ and $propV_{\ell'}.set$). Since any processor p_k in the intersection $propV_\ell.set \cap propV_{\ell'}.set$ has p_ℓ as its coordinator, $p_{\ell'}$ does not satisfy the condition $|\{p_k \in FD_{\ell'} : p_{\ell'} \in rep[k].FD \wedge rep[k].noCrd\}| > \lfloor n/2 \rfloor$ of line 9, and thus cannot propose a new view. Processor p_ℓ will install its view and remains the sole coordinator. Also, p_ℓ is the only one that can change its view due to failure detector change since it manages to get a majority of processors in $propV_\ell.set$ as opposed to $p_{\ell'}$.

Case 2(c): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Here we note that since p_ℓ has the greatest counter but has not included $p_{\ell'}$ to its $propV_\ell.set$, it should eventually be able to get all the processors in $propV_\ell.set$ to follow $propV_\ell$ by using the arguments of Case 2(a). In the mean time $p_{\ell'}$ will, in vain, be waiting for a response from p_ℓ accepting $propV_{\ell'}$. We note that $p_{\ell'}$ will not be able to initiate a new view once $propV_\ell$ is accepted, since it will not be able to gather a majority of processors with either $noCrd = \text{True}$ or proposed view $propV_{\ell'}$.

Case 2(d): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. This case is not symmetric to the above due to our assumption that p_ℓ is the one that has drawn the greatest view identifier from $inc()$. Here $propV_\ell.set$ includes $p_{\ell'}$ so p_ℓ waits for a response from $p_{\ell'}$ to proceed to the installation of $propV_\ell$. On the other hand, $p_{\ell'}$ will be waiting for responses from the processors in $propV_{\ell'}.set$. Any $p_i \in propV_\ell.set \cap propV_{\ell'}.set$ cannot keep $propV_\ell$ (even if initially it has accepted it, since it does not satisfy condition $p_{\ell'} \in rep[\ell].propV.set \Leftrightarrow p_\ell \in rep[\ell'].FD$ of line 6. Thus p_i accepts $propV_{\ell'}$ instead of $propV_\ell$, p_ℓ cannot propose a different view since it will not be able to get a majority of processors that have $propV_\ell$.

By the above exhaustive examination of cases, we reach to the result. Note that the above proof guarantees both convergence and closure of the algorithm to a legal execution, since p_ℓ remains the coordinator as long as it has a supporting majority. ■

Theorem 5.4 *Starting from an arbitrary configuration, any execution R of Algorithm 4 satisfying Definition 5.1, simulates automaton replication preserving the virtual synchrony property.*

Proof. We consider a finite prefix R' of R which has an arbitrary configuration c , and in which there exists a primary partition (as per Definition 5.1). Assume that this prefix is sufficiently long for Lemma 5.3 to hold, i.e., to reach a configuration c_{safe} in which there exists a global coordinator for a majority of processors. For this configuration we define a view v that has a coordinator p_ℓ and that any processor $p_i \in v$ that is not the coordinator is a *follower* of p_ℓ . We define a *multicast round* to be a sequence of ordered events: (i) *fetch()* input and propagate to coordinator, (ii) coordinator disseminates messages to be delivered in this new round, (iii) messages delivered and (iv) side effects produced by all processors. Our proof is broken into three steps that map the three possible transitions:

Step 1: Virtual synchrony is preserved between any two multicast rounds.

Suppose that there exists an input and a related message m in round r that is not delivered within r . We follow the multicast round r . First observe the following.

Remark: Within any multicast round, the coordinator executes lines 12 to 13 only once and a follower executes lines 16 to 18 only once, because the conditions are only satisfied the first time that the coordinator's local copy of the replica changes the round number.

By our Remark we notice that *fetch()* is called only once per round to collect input from the environment. This cannot be changed/overwritten since followers can never access $rep[i] \leftarrow rep[\ell]$ of line 17 that is the only line modifying the *input* field, unless they receive a new round number greater than the one they currently hold. We notice that the followers have produced side effects for the previous round (using *apply()*) based on the messages and state of the previous round. Similarly, the coordinator executes *fetch()* exactly once and only before it populates the *msg* array and after it has produced the side effects for the environment that were based on the previous messages (line 12). Line 13 populates the *msg* array with messages and including m . The coordinator p_ℓ then continuously propagates its current replica but cannot change it by the Remark and until condition $(\forall p_i \in v.set : rep_\ell[i].(view, status, rnd) = (view_\ell, status_\ell, rnd_\ell))$ (line 10) holds again. This ensures that the coordinator will change its *msg* array only when every follower has executed line 17 which allows the aforementioned condition to hold.

Any follower that keeps a previous round number does not allow the coordinator to move to the next round. If the coordinator moves to a new round, it is

implied that $rep[i] \leftarrow rep[\ell]$ and thus message m was received by any follower p_i , by our assumptions that the replica is propagated infinitely often and the data links are stabilizing. Thus, by the assumptions, any message m is certainly delivered within the view and round it was sent in, and thus the virtual synchrony property is preserved, whilst at the same time common state replication is achieved.

Step 2: Virtual synchrony is preserved in two consecutive view installations where there is no change of coordinator.

We now turn to the case where from one configuration c_{safe} we move to a new c'_{safe} that has a different view v' but has the same coordinator p_ℓ . Once p_ℓ is in an iteration where the condition $FD \neq propV.set$ of line 9 holds, a view change is required. Since p_ℓ is the global coordinator holds, no other processor can satisfy the condition ($|\{p_k \in FD_\ell : rep[k]_\ell.propV = propV_\ell\}| > \lfloor n/2 \rfloor$) of line 9, and so only p_ℓ . For more on why this holds one can refer to Lemma 5.2. Processor p_ℓ creates a new $propV$ with a new view ID taken from the increment counter algorithm, which is greater than the previous established view ID in $v.ID$. The last condition of line 10 guarantees that p_ℓ will not execute lines 12 to 14 and thus will not change its $rep.(state, input, msg)$ fields, until all the expected followers of the proposed view have sent their replicas. Followers that receive the proposal will accept it, since none of the conditions that existed change and so the new view proposal enforces that $valCrd = \{p_\ell\}$. Moreover, the proposal satisfies the condition of line 15 and the followers of the view enter status **Propose** leading to the installation of the view. What is important is that virtual synchrony is preserved since no follower is changing $rep.(state, input, msg)$ during this procedure, and moreover each sends its replica to p_ℓ by line 22. Once the replicas of all the followers have been collected, the coordinator creates a consolidated $state$ and msg array of all messages that were either delivered or pending. p_ℓ 's new replica is communicated to the followers who adopt this state as their own (line 19). Thus virtual synchrony is preserved and once all the processors have replicated the state of the coordinator, a new series of multicast rounds can begin by producing the side effects required by the input collected before the view change.

Step 3: Virtual synchrony is preserved in two consecutive view installations where the coordinator changes.

We assume that p_ℓ had a supporting majority throughout R' . We define a matching suffix R'' to prefix R' , such that R'' results from the loss of supporting majority by p_ℓ . Notice that since Definition 5.1 is required to hold, then some other processor with supporting majority $p_{\ell'}$, will by Lemma 5.2 propose the view v' with the highest view ID. We note that by the intersection property and the fact that a view set can only be formed by a majority set, $\exists p_i \in v \cap v'$. Thus, the “knowledge” of the system, $(state, input, msg)$ is retained within the majority.

As detailed in step 2, if a processor p_i had $noCrd = \text{True}$ for some time or was in status **Propose** it did not incur any changes to its replica. If it entered the **Install** phase, then this implies that the proposing processor has created a consolidated state that p_i has replicated. What is noteworthy is that whether

in status **Propose** or **Install**, if the proposer collapses (becomes inactive or suspected), the virtual synchrony property is preserved. It follows that, once status **Multicast** is reached by all followers, the system can start a practically infinite number of multicast rounds.

Thus, by the self-stabilization property of all the components of the system (counter increment algorithm, the data links, the failure detector and multicast) a legal execution is reached in which the virtual synchrony property is guaranteed and common state replication is preserved. ■

6 Conclusion

State-machine replication (SMR) is a service that simulates finite automata by letting the participating processors to periodically exchange messages about their current state as well as the last input that has led to this shared state. Thus, the processors can verify that they are in sync with each other. A well-known way to emulate SMRs is to use reliable multicast algorithms that guarantee *virtual synchrony* [4, 16]. To this respect, we have presented the first self-stabilizing algorithm that guarantees virtual synchrony, and used it to obtain a self-stabilizing SMR emulation; within this emulation, the system progresses in more extreme asynchronous executions in contrast to consensus-based SMRs, like the one in [9]. One of the key components of the virtual synchrony algorithm is a novel self-stabilizing counter algorithm, that establishes an efficient practical unbounded counter, which in turn can be directly used to implement a self-stabilizing MWMR register emulation; this extends the work in [1] that implements SWMR registers and can also be considered simpler and more communication efficient than the MWMR register implementation presented in [9].

Acknowledgements

We thank Iosif Salem for providing comments to improve the readability of this paper.

References

- [1] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sbastien Tixeuil. Practically stabilizing SWMR atomic memory in message-passing systems. *Journal of Computer and System Sciences*, 81(4):692 – 701, 2015.
- [2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability.
- [3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

- [4] Alberto Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.
- [5] Ken Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, pages 91–120, 2010.
- [6] Ken Birman and Robbert Van Renesse. *Reliable distributed computing with the Isis toolkit*. Wiley-IEEE Computer society press, Los Alamitos, 1994.
- [7] Ken Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
- [8] Kenneth P. Birman, Thomas A. Joseph, Thomas Ruchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Trans. Softw. Eng.*, 11(6):502–508, 1985.
- [9] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Practically self-stabilizing Paxos replicated state-machine. In *Proc. of the 2nd International Conference of Networked Systems (NETYS’14)*, pages 99–121, 2014.
- [10] Shlomi Dolev. *Self-Stabilization*, MIT press, 2000.
- [11] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.
- [12] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Proc. of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS’12)*, pages 133–147, 2012.
- [13] Shlomi Dolev, Ronen I. Kat, and Elad M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.
- [14] Shlomi Dolev, Limor Lahiani, Nancy A. Lynch, and Tina Nolte. Self-stabilizing mobile node location management and message routing. In *Self-Stabilizing Systems*, pages 96–112, 2005.
- [15] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, 2009.
- [16] Roger Khazan, Alan Fekete, and Nancy A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proc. of the 12th International Symposium on Distributed Computing (DISC’98)*, pages 258–272, 1998.

- [17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. pages 558–565, 1978. *Commun. ACM*, 21(7):558–565, 1978.
- [18] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of the 27th Annual International Symposium on Fault-Tolerant Computing (FTC 1997)*, pages 272–281, 1997.
- [19] Seth Gilbert, Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, 23(4):225–272, 2010.
- [20] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.